

全国计算机技术与软件专业技术资格（水平）考试指定用书

# 嵌入式系统设计师教程

魏洪兴 主编 谌卫军 康一梅 陈友东 编著

全国计算机技术与软件专业技术资格（水平）考试办公室组编

清华大学出版社



## 内 容 简 介

本书按照人事部、信息产业部全国计算机技术与软件专业技术资格(水平)考试要求编写,内容紧扣《嵌入式系统设计师考试大纲》。全书共6章,分别对嵌入式系统基础知识、嵌入式微处理器与接口设计、嵌入式软件与操作系统、嵌入式软件程序设计、嵌入式系统设计与维护等知识进行了详细的讲解,最后介绍了一个典型的嵌入式系统设计案例。

本书内容丰富,结构合理,概念清晰。既可作为全国计算机技术与软件专业技术资格(水平)考试中嵌入式系统设计师级别的考试用书,供有关考生学习使用,也可以作为本科生嵌入式系统相关课程教材或培训用书使用。

本书扉页为防伪页,封面贴有清华大学出版社防伪标签,无上述标识者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

### 图书在版编目(CIP)数据

嵌入式系统设计师教程 / 魏洪兴主编. —北京:清华大学出版社, 2006.8

(全国计算机技术与软件专业技术资格(水平)考试指定用书)

ISBN 978-7-302-13286-8

I. 嵌… II. 魏… III. 微型计算机-系统设计-工程技术人员-资格考核-教材 IV. TP360.21

中国版本图书馆 CIP 数据核字(2006)第 070690 号

责任编辑:柴文强 刘霞

责任印制:王秀菊

出版发行:清华大学出版社

地 址:北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编:100084

社 总 机:010-62770175

邮 购:010-62786544

投稿与读者服务:010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质 量 反 馈:010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 刷 者:北京密云胶印厂

装 订 者:三河市金元印装有限公司

经 销:全国新华书店

开 本:185×230 印张:33.75 字数:753 千字

版 次:2006 年 8 月第 1 版 印 次:2010 年 2 月第 5 次印刷

印 数:14501~17500

定 价:50.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话:(010)62770177 转 3103 产品编号:022594-01/TP



# 序

在国务院鼓励软件产业发展政策的带动下，我国软件业一年一大步，实现了跨越式发展，相关政策措施正在落实，我国软件产业的国际竞争力日益提高。

在软件产业快速发展的带动下，人才需求日益迫切，队伍建设与时俱进，而作为规范软件专业技术人员技术资格的计算机软件考试已在我国实施了十余年，累计报考人数超过一百五十万，为推动我国软件产业的发展作出了重要贡献。

软件考试在全国率先执行了以考代评的政策，取得了良好的效果。为贯彻落实国务院颁布的《振兴软件产业行动纲要》和国家职业资格证书制度，国家人事部和信息产业部对计算机软件考试政策进行了重大改革：考试名称调整为计算机技术与软件专业技术资格（水平）考试；考试对象从狭义的计算机软件扩大到广义的计算机软件，涵盖了计算机技术与软件的各个主要领域（5个专业类别、3个级别层次和20个职业岗位资格）；资格考试和水平考试合并，采用水平考试的形式（与国际接轨，报考不限学历与资历条件），执行资格考试政策（各用人单位可以从考试合格者中择优聘任专业技术职务）；这是我国人事制度改革的一次新突破。此外，将资格考试政策延伸到高级资格，使考试制度更为完善。

信息技术发展快，更新快，要求从业人员不断适应和跟进技术的变化，有鉴于此，国家人事部和信息产业部规定对通过考试获得的资格（水平）证书实行每隔三年进行登记的制度，以鼓励和促进专业人员不断接受新知识、新技术、新法规的继续教育。考试设置的专业类别、职业岗位也将随着国民经济与社会发展而动态调整。

目前，我国计算机软件考试的部分级别已与日本和韩国信息处理工程师考试的相应级别实现了互认，以后还将继续扩大考试互认的级别和国家。

为规范培训和考试工作，信息产业部电子教育中心组织一批具有较高理论水平和丰富实践经验的专家编写了全国计算机技术与软件专业技术资格（水平）考试的教材和辅导用书，按照考试大纲的要求，全面介绍相关知识与技术，帮助考生学习和备考。

我们相信，经过全社会的共同努力，全国计算机技术与软件专业技术资格（水平）考试将会更加规范、科学，进而对培养信息技术人才，加快专业队伍建设，推动国民经济和社会信息化作出更大的贡献。

信息产业部副部长 姜勤俭

# 前 言

全国计算机技术与软件专业技术资格（水平）考试实施至今已经历了近 20 年，在社会上产生了很大的影响，对我国软件产业的形成和发展做出了重要的贡献。为了适应我国计算机信息技术发展的需求，国家人事部和信息产业部决定将考试的级别拓展到计算机信息技术行业的各个方面，以满足社会上对各种计算机信息技术人才的需要。

以数字科技（微电子是其重要组成部分）为基础，计算机科学技术为框架的嵌入式系统目前已普遍应用于工业控制系统、信息家电、通信设备、医疗仪器、智能仪器仪表等众多领域，如手机、PDA、MP3、手持设备、智能电话、机顶盒等，可以说嵌入式技术无处不在。由于社会对掌握嵌入式技术人才的大量需求，使嵌入式软硬件工程师成为未来几年的热门职业之一。为了推动国内嵌入式技术的发展和嵌入式人才培养，全国计算机技术与软件专业技术资格（水平）考试设立了“嵌入式系统设计师”级别考试内容。

编者受全国计算机技术与软件专业技术资格（水平）考试办公室委托，编写《嵌入式系统设计师教程》一书，以适应嵌入式系统设计师级别的考试大纲要求。由于嵌入式系统设计涉及计算机基础知识、数字逻辑电路基础、微处理器原理与接口技术、实时操作系统、嵌入式软件编程等诸多知识内容，编写相应的教程难度较大。编者在撰写本书时紧扣《嵌入式系统设计师考试大纲》，对考生需掌握的内容进行了全面、深入的阐述。考虑到参加考试的人员已有一定的基础，所以本书中只对考试大纲中所涉及到的重点知识领域加以阐述，限于篇幅不能详细地展开，请读者谅解。

全书共分 6 章，第 1 章由魏洪兴、贾智平、陈友东编写，第 2 章由魏洪兴编写，第 3 章和第 4 章由谌卫军编写，第 5 章由康一梅、陈友东编写，第 6 章由陈友东编写，全书由魏洪兴统稿。

在本书的编写过程中，参考了许多相关的书籍和资料，编者在此对这些参考文献的作者表示感谢。同时感谢清华大学出版社在本书出版过程中所给予的支持和帮助。

因水平有限，书中难免存在错漏和不妥之处，望读者指正，以利改进和提高。

编 者

2006 年 4 月



# 目 录

第 1 章 嵌入式系统基础知识.....1	第 2 章 嵌入式微处理器与接口知识 .....83
1.1 嵌入式系统的定义和组成.....1	2.1 嵌入式微处理器的结构和类型 .....83
1.1.1 嵌入式系统的定义.....1	2.1.1 嵌入式微处理器的分类 .....83
1.1.2 嵌入式系统发展概述.....2	2.1.2 典型 8 位微处理器的结构 和特点 .....86
1.1.3 嵌入式系统的组成.....5	2.1.3 典型 16 位微处理器的结构 和特点 .....94
1.1.4 实时系统.....12	2.1.4 典型 32 位微处理器的结构 和特点 .....97
1.2 嵌入式微处理器体系结构.....17	2.1.5 DSP 处理器的结构和特点 .....112
1.2.1 冯·诺依曼与哈佛结构.....17	2.1.6 多核处理器的结构和特点 .....118
1.2.2 CISC 与 RISC.....18	2.2 嵌入式系统的存储体系 .....124
1.2.3 流水线技术.....23	2.2.1 存储器系统概述 .....124
1.2.4 信息存储的字节顺序.....25	2.2.2 嵌入式系统存储设备分类 .....132
1.3 嵌入式系统的硬件基础.....28	2.2.3 ROM 的种类与选型 .....135
1.3.1 组合逻辑电路基础.....28	2.2.4 Flash Memory 的种类与 选型 .....137
1.3.2 时序逻辑电路.....35	2.2.5 RAM 的种类与选型 .....141
1.3.3 总线电路及信号驱动.....39	2.2.6 外部存储器的种类与选型 .....145
1.3.4 电平转换电路.....47	2.3 嵌入式系统输入输出设备 .....151
1.3.5 可编程逻辑器件基础.....51	2.3.1 嵌入式系统常用输入/输出 设备概述 .....151
1.4 嵌入式系统中信息表示与运算基础.....61	2.3.2 GPIO 原理与结构.....153
1.4.1 进位计数制与转换.....61	2.3.3 A/D 接口基本原理与结构 .....154
1.4.2 计算机中数的表示.....62	2.3.4 D/A 接口基本原理与结构 .....159
1.4.3 非数值数据编码.....65	2.3.5 键盘接口基本原理与结构 .....161
1.4.4 差错控制编码.....70	2.3.6 显示接口基本原理与结构 .....164
1.5 嵌入式系统的性能评价.....77	
1.5.1 度量项目.....77	
1.5.2 评价方法.....79	
1.5.3 评估嵌入式系统处理器的主 要指标.....81	

2.3.7 触摸屏接口基本原理与结构.....173	3.1.1 嵌入式软件概述.....246
2.3.8 音频接口基本原理与结构.....177	3.1.2 嵌入式软件分类.....247
2.4 嵌入式系统总线接口.....179	3.1.3 嵌入式软件的体系结构.....247
2.4.1 串行接口基本原理与结构.....179	3.1.4 设备驱动层.....251
2.4.2 并行接口基本原理与结构.....185	3.1.5 嵌入式中间件.....253
2.4.3 PCI 接口基本原理与结构.....187	3.2 嵌入式操作系统概述.....254
2.4.4 USB 接口基本原理与结构.....189	3.2.1 嵌入式操作系统的概念.....254
2.4.5 SPI 接口基本原理与结构.....193	3.2.2 嵌入式操作系统的分类.....255
2.4.6 IIC 接口基本原理与结构.....195	3.2.3 常见的嵌入式操作系统.....258
2.4.7 PCMCIA 接口基本原理与结构.....198	3.3 任务管理.....261
2.5 嵌入式系统网络接口.....199	3.3.1 多道程序技术.....261
2.5.1 以太网接口基本原理与结构.....199	3.3.2 进程、线程和任务.....262
2.5.2 CAN 总线接口的基本原理与结构.....204	3.3.3 任务的实现.....270
2.5.3 xDSL 接口基本原理与结构.....209	3.3.4 任务的调度.....277
2.5.4 无线以太网基本原理与结构.....213	3.3.5 实时系统调度.....286
2.5.5 蓝牙接口基本原理与结构.....215	3.3.6 任务间的同步与互斥.....290
2.5.6 1394 接口基本原理与结构.....219	3.3.7 任务间通信.....299
2.6 嵌入式系统电源.....222	3.4 存储管理.....301
2.6.1 电源接口技术.....222	3.4.1 存储管理概述.....301
2.6.2 电源管理技术.....223	3.4.2 实模式与保护模式.....302
2.7 电子电路设计基础.....227	3.4.3 分区存储管理.....305
2.7.1 电路设计.....227	3.4.4 地址映射.....311
2.7.2 PCB 电路设计.....229	3.4.5 页式存储管理.....315
2.7.3 电子设计.....236	3.4.6 虚拟存储管理.....321
2.7.4 电子电路测试.....241	3.5 设备管理.....328
第3章 嵌入式系统软件及操作 系统知识.....246	3.5.1 设备管理基础.....328
3.1 嵌入式软件基础.....246	3.5.2 I/O 控制方式.....329
	3.5.3 I/O 软件.....332
	3.6 文件系统.....335
	3.6.1 嵌入式文件系统概述.....335
	3.6.2 文件和目录.....336
	3.6.3 文件系统的实现.....338
第4章 嵌入式软件程序设计.....343	
4.1 嵌入式软件开发概述.....343	

4.1.1 嵌入式应用开发过程	343	5.2.1 系统分析的目的和任务	435
4.1.2 嵌入式软件开发的特点	343	5.2.2 用户需求	436
4.1.3 嵌入式软件开发的挑战	345	5.2.3 系统需求	438
4.2 嵌入式程序设计语言	346	5.2.4 系统规格说明书的编写方法	443
4.2.1 程序设计语言概述	346	5.3 系统设计知识	447
4.2.2 汇编语言	351	5.3.1 传统的系统设计方法	447
4.2.3 面向过程的语言	355	5.3.2 实时系统分析与设计	449
4.2.4 面向对象的语言	362	5.3.3 软硬件协同设计方法	452
4.2.5 汇编、编译与解释程序的基本原理	367	5.4 系统实施知识	461
4.3 嵌入式软件开发环境	374	5.4.1 系统架构设计	461
4.3.1 宿主机、目标机	375	5.4.2 系统详细设计	472
4.3.2 嵌入式软件开发工具	376	5.4.3 系统测试	475
4.3.3 集成开发环境	384	5.5 系统维护知识	480
4.4 嵌入式软件开发	388	5.5.1 系统运行管理	480
4.4.1 嵌入式平台选型	388	5.5.2 系统维护知识	484
4.4.2 软件设计	390	5.5.3 系统评价知识	487
4.4.3 嵌入式程序设计	396		
4.4.4 编码	399		
4.4.5 测试	402		
4.4.6 下载和运行	410		
4.5 嵌入式软件移植	412		
4.5.1 无操作系统的软件移植	412		
4.5.2 有操作系统的软件移植	413		
4.5.3 应用软件的移植	415		
第 5 章 嵌入式系统开发与维护知识	417		
5.1 系统开发过程及其项目管理	417		
5.1.1 系统开发生命周期各阶段的目标和任务的划分方法	417		
5.1.2 系统开发项目管理基础知识及其常用管理工具使用方法	421		
5.1.3 系统开发工具与环境知识	425		
5.2 系统分析基础知识	435		
		第 6 章 嵌入式系统设计	490
		6.1 嵌入式系统设计的特点	490
		6.2 嵌入式系统的设计流程	492
		6.2.1 产品定义	494
		6.2.2 嵌入式系统的软硬件划分	495
		6.2.3 嵌入式系统硬件设计	496
		6.2.4 嵌入式系统的软件设计	498
		6.2.5 系统集成和测试	503
		6.3 设计示例：嵌入式数控系统	503
		6.3.1 数控系统简介	505
		6.3.2 需求分析	505
		6.3.3 系统体系结构设计	506
		6.3.4 硬件设计	509
		6.3.5 软件设计	521
		6.3.6 系统集成与测试	527



# 第 1 章 嵌入式系统基础知识

## 1.1 嵌入式系统的定义和组成

### 1.1.1 嵌入式系统的定义

嵌入式系统是一种应用范围非常广泛的系统。可以说除了桌面计算机和服务器外所有计算设备都属于嵌入式系统，例如从便携式音乐播放器到航天飞机上的实时系统控制都属于嵌入式系统。

大多数商用的嵌入式系统都设计成专用任务的低成本的产品。大多数的嵌入式系统都具有实时性的要求。有些功能需要非常快的主频，但其他大多数功能并不需要高速的处理能力。这些系统通过特定的器件和软件来满足实时性的要求。

简单地通过速度和成本来定义嵌入式系统是困难的，但对于大批量的产品而言，成本常常对系统设计起决定作用。通常，一个嵌入式系统的很多部分相对系统主要功能来说需要较低的性能，因此嵌入式系统和通用 PC 相比，能够使用一个满足辅助功能的合适的 CPU，从而简化了系统设计，降低了成本。例如，数字电视的机顶盒需要处理每秒以百万兆位计的连续数据，但这些数据处理大部分是由定制的硬件来实现的，如解析、管理和编解码多个频道的数字影像。

对于大批量生产的嵌入式系统，如便携式音乐播放器或手机等，降低成本就成为最主要的问题。这些系统通常只具有几个芯片：一个高度集成的 CPU，一个定制的芯片用于控制其他所有的功能，还有一个存储芯片。在这种设计中，每部分都设计成使用最小的系统功耗。

对于小批量的嵌入式应用，为了降低开发成本，常常使用 PC 体系结构，通过限制程序的执行时间或用实时操作系统来替换原先的操作系统。在这种情况下，可以使用一个或多个高性能的 CPU 来替换特殊用途的硬件。

嵌入式系统的软件通常运行在有限的硬件资源上：没有硬盘、操作系统、键盘或屏幕。软件一般都没有文件系统，如果有的话，也会采用 Flash 驱动器。如果有人机交互接口的话，也是一个小键盘或液晶显示器。硬件是计算机的物理部分，和存储在硬件中的计算机



软件程序和数据区分开来。

嵌入到机械中的嵌入式系统需要长期无故障连续运行，因此它的软件需要比 PC 中的软件更加仔细地开发和更加严格地测试。

那么，到底什么是嵌入式系统呢？

根据 IEEE（国际电气和电子工程师协会）的定义，嵌入式系统是“控制、监视或者辅助设备、机器和车间运行的装置”（原文为 *devices used to control, monitor, or assist the operation of equipment, machinery or plants*）。这主要是从应用上加以定义的，从中可以看出嵌入式系统是软件和硬件的综合体，还可以涵盖机械等附属装置。

目前国内一个普遍被认同的定义是：以应用为中心、以计算机技术为基础，软件硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。

可以这样认为，嵌入式系统是一种专用的计算机系统，作为装置或设备的一部分。通常，嵌入式系统是一个控制程序存储在 ROM 中的嵌入式处理器控制板。事实上，所有带有数字接口的设备，如手表、微波炉、录像机、汽车等，都使用嵌入式系统，有些嵌入式系统还包含操作系统，但大多数嵌入式系统都是由单个程序实现整个控制逻辑。

### 1.1.2 嵌入式系统发展概述

#### 1. 嵌入式系统的发展历史

世界上第一个应用的嵌入式系统可以追溯到 20 世纪 60 年代中期的阿波罗导航计算机 AGC（Apollo Guidance Computer）系统，用来完成阿波罗飞船的导航控制。当时随着电子技术的发展，嵌入式计算机才逐步兴起。从单片机的出现到今天各种嵌入式微处理器、微控制器的广泛应用，嵌入式系统少说也有了 30 多年的历史。纵观嵌入式系统的发展历程，大致经历了以下 4 个阶段。

##### （1）无操作系统阶段

嵌入式系统最初的应用是基于单片机的，大多以可编程控制器的形式出现，具有监测、伺服和设备指示等功能，通常应用于各类工业控制和飞机、导弹等武器装备中，一般没有操作系统的支持，只能通过汇编语言对系统进行直接控制，运行结束后再清除内存。这些装置虽然已经初步具备了嵌入式的应用特点，但仅仅是使用 8 位的 CPU 芯片来执行一些单线程的程序，因此严格地说还谈不上“系统”的概念。

这一阶段嵌入式系统的主要特点是：系统结构和功能相对单一，处理效率较低，存储容量较小，几乎没有用户接口。由于这种嵌入式系统使用简便、价格低廉，因而曾经在工业控制领域中得到了非常广泛的应用，但却无法满足现今对执行效率和存储容量都有较高要求的信息家电等的需要。

## (2) 简单操作系统阶段

20 世纪 80 年代, 随着微电子工艺水平的提高, 集成电路 (Integrated Circuit, IC) 制造商开始把嵌入式应用中所需要的微处理器、I/O 接口、串行接口及 RAM、ROM 等部件集成到一片 VLSI 中, 制造出面向 I/O 设计的微控制器, 并一举成为嵌入式系统领域中异军突起的新秀。与此同时, 嵌入式系统的程序员也开始基于一些简单的“操作系统”开发嵌入式应用软件, 大大缩短了开发周期、提高了开发效率。

这一阶段嵌入式系统的主要特点是: 出现了大量具有高可靠性、低功耗的嵌入式 CPU (如 Power PC 等), 各种简单的嵌入式操作系统开始出现并得到迅速发展。此时的嵌入式操作系统虽然还比较简单, 但已经初步具有了一定的兼容性和扩展性, 内核精巧且效率高, 主要用来控制系统负载及监控应用程序的运行。

## (3) 实时操作系统阶段

20 世纪 90 年代, 在分布控制、柔性制造、数字化通信和信息家电等巨大需求的牵引下, 嵌入式系统进一步飞速发展, 而面向实时信号处理算法的数字信号处理器 (Digital Signal Processor, DSP) 产品则向着高速度、高精度、低功耗的方向发展。随着硬件实时性要求的提高, 嵌入式系统的软件规模也不断扩大, 逐渐形成了实时多任务操作系统 (Real-time Operation System, RTOS), 并开始成为嵌入式系统的主流。

这一阶段嵌入式系统的主要特点是: 操作系统的实时性得到了很大改善, 已经能够运行在各种不同类型的微处理器上, 具有高度的模块化和扩展性。此时的嵌入式操作系统已经具备了文件和目录管理、设备管理、多任务、网络、图形用户界面 (Graphic User Interface, GUI) 等功能, 并提供了大量的应用程序接口 (Application Programming Interface, API), 从而使应用软件的开发变得更加简单。

## (4) 面向 Internet 阶段

21 世纪是网络时代, 随着 Internet 的进一步发展, 以及 Internet 技术与信息家电、工业控制技术等的结合日益紧密, 嵌入式设备与 Internet 的结合是嵌入式系统未来的发展趋势。

## 2. 嵌入式系统的发展趋势

信息时代和数字时代的到来, 为嵌入式系统的发展带来了巨大的机遇, 同时也向嵌入式系统厂商提出了新的挑战。目前, 嵌入式技术与 Internet 技术的结合正在推动着嵌入式系统的飞速发展, 嵌入式系统的研究和应用产生了如下新的显著变化:

- 新的微处理器层出不穷, 嵌入式操作系统自身结构的设计更加便于移植, 能够在短时间内支持更多的微处理器。
- 嵌入式系统的开发成了一项系统工程, 开发厂商不仅要提供嵌入式软硬件系统本身, 同时还要提供强大的硬件开发工具和软件支持包。
- 通用计算机上使用的新技术、新观念开始逐步移植到嵌入式系统中, 嵌入式软件



平台得到进一步完善。

- 各类嵌入式 Linux 操作系统迅速发展, 由于具有源代码开放、系统内核小、执行效率高、网络结构完整等特点, 很适合信息家电等嵌入式系统的需要, 目前已经形成了能与 Windows CE、Palm OS 等嵌入式操作系统进行有力竞争的局面。
- 网络化、信息化的要求随着 Internet 技术的成熟和带宽的提高而日益突出, 以往功能单一的设备(如电话、手机、冰箱、微波炉等)功能不再单一, 结构变得更加复杂, 网络互联成为必然趋势。
- 精简系统内核, 优化关键算法, 降低功耗和软硬件成本。
- 提供更加友好的多媒体人机交互界面。

### 3. 知识产权核

IC 产业是一个自 20 世纪 80 年代特别是 90 年代后飞速发展的产业。从 90 年代中期开始, 由于基于专用集成电路的板级系统设计已经不能满足系统产品的可靠性等要求, 出现了片上系统(System On Chip, SOC)的概念, 并成为现代集成电路设计的发展方向。SOC 是指在单芯片上集成数字信号处理器、微控制器、存储器、数据转换器、接口电路等电路模块, 可以直接实现信号采集、转换、存储、处理等功能, 其中知识产权核(Intellectual Property Core, IP Core)设计是 SOC 设计的基础。

IP 核是指具有知识产权的、功能具体、接口规范、可在多个集成电路设计中重复使用的功能模块, 是实现系统芯片(SOC)的基本构件。IP 核在功能设计上考虑了可重用性, 验证方法也非常明确。IP 核模块有行为(Behavior)、结构(Structure)和物理(Physical)3 级不同程度的设计, 对应描述功能行为的不同分为三类, 即软核(Soft IP Core)、完成结构描述的固核(Firm IP Core)和基于物理描述并经过工艺验证的硬核(Hard IP Core)。

- IP 软核(Soft IP Core): 通常是用硬件描述语言(hardware Description Language, HDL)文本形式提交给用户, 它经过 RTL 级设计优化和功能验证, 但其中不含有任何具体的物理信息。据此, 用户可以综合出正确的门电路级设计网表, 并可以进行后续的结构设计, 具有很大的灵活性, 借助于 EDA 综合工具可以很容易地与其他外部逻辑电路合成一体, 根据各种不同半导体工艺, 设计成具有不同性能的器件。其主要缺点是缺乏对时序、面积和功耗的预见性。而且 IP 软核以源代码的形式提供的, IP 知识产权不易保护。
- IP 硬核(Hard IP Core)是基于半导体工艺的物理设计, 已有固定的拓扑布局 and 具体工艺, 并已经过工艺验证, 具有可保证的性能。其提供给用户的形式是电路物理结构掩模版图和全套工艺文件。由于无需提供寄存器转移级(Register transfer level, RTL)文件, 因而更易于实现 IP 保护。其缺点是灵活性和可移植性差。
- IP 固核(Firm IP Core)的设计程度则是介于软核和硬核之间, 除了完成软核所有

的设计外，还完成了门级电路综合和时序仿真等设计环节。一般以门级电路网表的形式提供给用户。

IC 设计中采用 IP 复用可以缩短产品的开发周期，提高产品的可靠性。全球 IP 核市场目前处于快速成长的阶段，1999 年到 2004 年的增长率高达 43%。2001 年全球 IP 核市场规模达 8.9 亿美元，较 2000 年的 7.1 亿美元增长了 25%。在十大 IP 供应商排行中，ARM、Rambus 和 MIPS 居前 3 位。

为了保护 IP 核的开发与使用者，同时建立良好的 IP 核技术基础，全球各界已筹备了许多策略联盟，如 EDA 联盟、RAPID 联盟、VCX 联盟与 VSIA 联盟等，来积极推动 IP 核的开发、应用及推广。其中 EDA 联盟主要由提供集成电路自动化设计软件的公司所组成，主要工作是要提升集成电路设计产业对 EDA 软件功能的认知与肯定，同时建立 EDA 公司与集成电路设计公司沟通交流渠道与解决集成电路产业所面临的问题，所以 EDA 联盟主要是以如何提供更好的 EDA 软件工具为主，也处理一部分 IP 核使用标准的问题。而 VSIA 联盟主要是针对 IP 核可复用性进行规范，希望建立一个共性标准，以方便实现将不同公司的 IP 核整合于一个 SOC 芯片中。VSIA 联盟针对 IP 核的定义、开发、授权及测试等建立了一个公开的共性规范。

### 1.1.3 嵌入式系统的组成

嵌入式系统是一种应用范围非常广泛的系统。可以说除了一般用途的计算机外的所有计算机都属于嵌入式系统，例如从便携式音乐播放器到航天飞机上的实时系统控制都属于嵌入式系统。

大多数商用的嵌入式系统都设计成专用任务的低成本的产品。大多数的嵌入式系统都具有实时性的要求。有些功能需要非常快的主频，但其他大多数功能并不需要高速的处理能力。这些系统通过特定的器件和软件来满足实时性的要求。

简单地通过速度和成本来定义嵌入式系统是困难的，但对于大批量的产品而言，成本常常对系统设计起决定作用。通常，一个嵌入式系统的很多部分相对于系统主要功能来说需要较低的性能，因此嵌入式系统和通用 PC 相比，能够使用一个可满足辅助功能的合适的 CPU，从而简化了系统设计，降低了成本。例如，数字电视的机顶盒需要处理每秒以百万兆位计的连续数据，但这些数据处理大部分是由定制的硬件来实现的，如解析、管理和编解码多个频道的数字影像。

对于大批量生产的嵌入式系统，如便携式音乐播放器或手机等，降低成本就成为最主要的问题。这些系统通常只具有几个芯片：一个高度集成的 CPU，一个定制的芯片用于控制其他所有的功能，还有一个存储芯片。在这种设计中，每部分都设计成使用最小的系统功耗。

对于小批量的嵌入式应用，为了降低开发成本，常常使用 PC 体系结构，通过限制程序的执行时间或用实时操作系统来替换原先的操作系统。在这种情况下，可以使用一个或多个高性能的 CPU 来替换特殊用途的硬件。

嵌入式系统的软件通常运行在有限的硬件资源上：没有硬盘、操作系统、键盘或屏幕。软件一般都没有文件系统，如果有，也会采用 Flash 驱动器。如果有人机交互接口，也是一个小键盘或液晶显示器。硬件是计算机的物理部分，和存储在硬件中的计算机软件程序及数据区分开来。

嵌入到机械中的嵌入式系统需要长期无故障连续运行，因此它的软件需要比 PC 中的软件更加仔细的开发和更加严格的测试。

那么，到底什么是嵌入式系统呢？

### 1. 嵌入式系统的定义

根据 IEEE（国际电气和电子工程师协会）的定义，嵌入式系统是“控制、监视或者辅助设备、机器和车间运行的装置”。这主要是从应用上加以定义的，从中可以看出嵌入式系统是软件和硬件的综合体，还可以涵盖机械等附属装置。

目前国内一个普遍被认同的定义是：以应用为中心、以计算机技术为基础，软件硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗严格要求的专用计算机系统。

可以这样认为，嵌入式系统是一种专用的计算机系统，作为装置或设备的一部分。通常，嵌入式系统是一个控制程序存储在 ROM 中的嵌入式处理器控制板。事实上，所有带有数字接口的设备，如手表、微波炉、录像机、汽车等，都使用嵌入式系统，有些嵌入式系统还包含操作系统，但大多数嵌入式系统都是由单个程序实现整个控制逻辑。

### 2. 嵌入式系统的组成

一个嵌入式系统装置一般都由嵌入式计算机系统和执行装置组成，如图 1-1 所示，嵌入式计算机系统是整个嵌入式系统的核心，由硬件层、中间层、系统软件层和应用软件层组成。执行装置也称为被控对象，它可以接受嵌入式计算机系统发出的控制命令，执行所规定的操作或任务。执行装置可以很简单，如手机上的一个微小型的电机，当手机处于震动接收状态时打开；也可以很复杂，如 SONY 智能机器狗，上面集成了多个微小型控制电机和多种传感器，从而可以执行各种复杂的动作和感受各种状态信息。

下面对嵌入式计算机系统的组成进行介绍。

#### 1) 硬件层

硬件层中包含嵌入式微处理器、存储器（SDRAM、ROM、Flash 等）、通用设备接口和 I/O 接口（A/D、D/A、I/O 等）。在一片嵌入式处理器基础上添加电源电路、时钟电路和存储器电路，就构成了一个嵌入式核心控制模块。其中操作系统和应用程序都可以固化在 ROM 中。



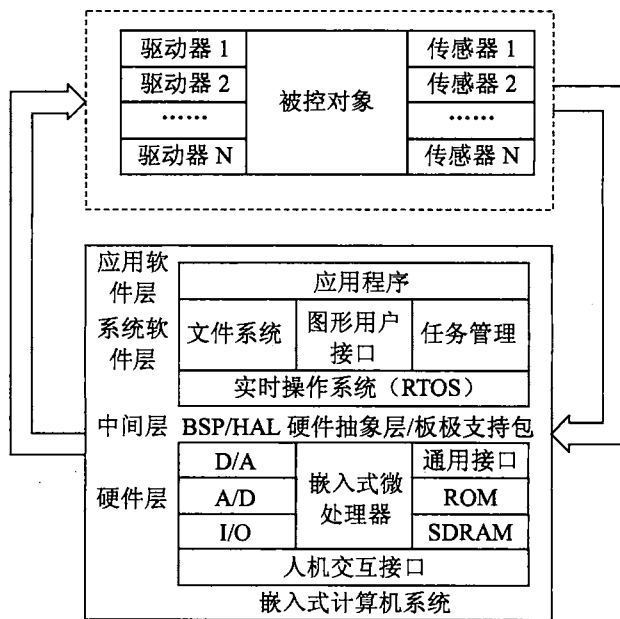


图 1-1 嵌入式系统的典型组成

### (1) 嵌入式微处理器

嵌入式系统硬件层的核心是嵌入式微处理器，嵌入式微处理器与通用 CPU 最大的不同在于嵌入式微处理器大多工作在为特定用户群所专门设计的系统中；它将通用 CPU 中许多由板卡完成的任务集成到芯片内部，从而有利于嵌入式系统在设计时趋于小型化，同时还具有很高的效率和可靠性。

嵌入式微处理器的体系结构可以采用冯·诺依曼体系结构或哈佛体系结构；指令系统可以选用精简指令系统（Reduced Instruction Set Computer, RISC）和复杂指令集系统 CISC（Complex Instruction Set Computer, CISC）。CISC 计算机具有大量的指令和寻址方式，但大多数程序只使用少量的指令就能够运行；RISC 计算机在通道中只包含最有用的指令，确保数据通道快速执行每一条指令，从而提高了执行效率并使 CPU 硬件结构设计变得更为简单。

嵌入式微处理器有各种不同的体系，即使在同一体系中也可能具有不同的时钟频率和数据总线宽度，或集成了不同的外设和接口。据不完全统计，目前全世界嵌入式微处理器已经超过 1000 多种，体系结构有 30 多个系列，其中主流的体系有 ARM、MIPS、PowerPC、X86 和 SH 等。但与全球 PC 市场不同的是，没有一种嵌入式微处理器可以主导市场，仅以 32 位的产品而言，就有 100 种以上的嵌入式微处理器。嵌入式微处理器的选择是根据具体

的应用而决定的。

## (2) 存储器

嵌入式系统需要存储器来存放和执行代码。嵌入式系统的存储器包含 Cache、主存和辅助存储器，其存储结构如图 1-2 所示。

### ① Cache

Cache 是一种容量小、速度快的存储器阵列，它位于主存和嵌入式微处理器内核之间，存放的是最近一段时间微处理器使用最多的程序代码和数据。在需要进行数据读取操作时，微处理器尽可能的从 Cache 中读取数据，而不是从主存中读取，这样就大大改善了系统的性能，提高了微处理器和主存之间的数据传输速率。Cache 的主要目标就是：减小存储器（如主存和辅助存储器）给微处理器内核造成的存储器访问瓶颈，使处理速度更快，实时性更强。

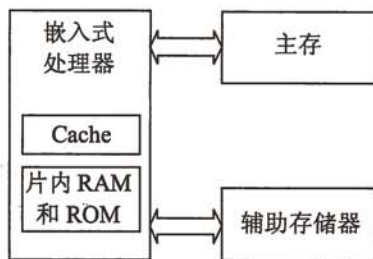


图 1-2 嵌入式系统的存储结构

在嵌入式系统中 Cache 全部集成在嵌入式微处理器内，可分为数据 Cache、指令 Cache 或混合 Cache，Cache 的大小依不同处理器而定。一般中高档的嵌入式微处理器才会把 Cache 集成进去。

### ② 主存

主存是嵌入式微处理器能直接访问的寄存器，用来存放系统和用户的程序及数据。它可以位于微处理器的内部或外部，其容量为 256KB~1GB，根据具体的应用而定，一般片内存储器容量小，速度快，片外存储器容量大。

常用作主存的存储器有：

- ROM 类 NOR Flash、EPROM 和 PROM 等。
- RAM 类 SRAM、DRAM 和 SDRAM 等。

其中 NOR Flash 凭借其可擦写次数多、存储速度快、存储容量大、价格便宜等优点，在嵌入式领域内得到了广泛应用。

### ③ 辅助存储器

辅助存储器用来存放大数据量的程序代码或信息，它的容量大，但读取速度与主存相比就慢得多，用来长期保存用户的信息。

嵌入式系统中常用的外存有：硬盘、NAND Flash、CF 卡、MMC 和 SD 卡等。

## (3) 通用设备接口和 I/O 接口

嵌入式系统和外界交互需要一定形式的通用设备接口，如 A/D、D/A、I/O 等，外设通过和片外其他设备的或传感器的连接来实现微处理器的输入/输出功能。每个外设通常都只有单一的功能，它可以在芯片外也可以内置在芯片中。外设的种类很多，可从一个简单的

串行通信设备到非常复杂的 802.11 无线设备。

目前嵌入式系统中常用的通用设备接口有 A/D（模/数转换接口）、D/A（数/模转换接口）、I/O 接口有 RS-232 接口（串行通信接口）、Ethernet（以太网接口）、USB（通用串行总线接口）、音频接口、VGA 视频输出接口、I2C（现场总线）、SPI（串行外围设备接口）和 IrDA（红外线接口）等。

## 2) 中间层

硬件层和软件层之间为中间层，也称为硬件抽象层（Hardware Abstract Layer, HAL）或板级支持包（Board Support Package, BSP），它将系统上层软件与底层硬件分离开来，使系统的底层驱动程序与硬件无关，上层软件开发人员无需关心底层硬件的具体情况，根据 BSP 层提供的接口即可进行开发。该层一般包含相关底层硬件的初始化、数据的输入/输出操作和硬件设备的配置等功能。BSP 具有以下两个特点。

- 硬件相关性：因为嵌入式实时系统的硬件环境具有应用相关性，而作为上层软件与硬件平台之间的接口，BSP 需要为操作系统提供操作和控制具体硬件的方法。
- 操作系统相关性：不同的操作系统具有各自的软件层次结构，因此，不同的操作系统具有特定的硬件接口形式。

在实现上，BSP 是一个介于操作系统和底层硬件之间的软件层次，包括了系统中大部分与硬件联系紧密的软件模块。设计一个完整的 BSP 需要完成两部分工作：嵌入式系统的硬件初始化以及 BSP 功能，设计硬件相关的设备驱动。

### (1) 嵌入式系统硬件初始化

系统初始化过程可以分为 3 个主要环节，按照自底向上、从硬件到软件的次序依次为：片级初始化、板级初始化和系统级初始化。

#### • 片级初始化

完成嵌入式微处理器的初始化，包括设置嵌入式微处理器的核心寄存器和控制寄存器、嵌入式微处理器核心工作模式和嵌入式微处理器的局部总线模式等。片级初始化把嵌入式微处理器从上电时的默认状态逐步设置成系统所要求的工作状态。这是一个纯硬件的初始化过程。

#### • 板级初始化

完成嵌入式微处理器以外的其他硬件设备的初始化。另外，还需设置某些软件的数据结构和参数，为随后的系统级初始化和应用程序的运行建立硬件和软件环境。这是一个同时包含软硬件两部分在内的初始化过程。

#### • 系统级初始化

该初始化过程以软件初始化为主，主要进行操作系统的初始化。BSP 将对嵌入式微处理器的控制权转交给嵌入式操作系统，由操作系统完成余下的初始化操作，包含加载和初



始化与硬件无关的设备驱动程序，建立系统内存区，加载并初始化其他系统软件模块，如网络系统、文件系统等。最后，操作系统创建应用程序环境，并将控制权交给应用程序的入口。

### (2) 硬件相关的设备驱动程序

BSP 的另一个主要功能是硬件相关的设备驱动。硬件相关的设备驱动程序的初始化通常是一个从高到低的过程。尽管 BSP 中包含硬件相关的设备驱动程序，但是这些设备驱动程序通常不直接由 BSP 使用，而是在系统初始化过程中由 BSP 将他们与操作系统中通用的设备驱动程序关联起来，并在随后的应用中由通用的设备驱动程序调用，实现对硬件设备的操作。与硬件相关的驱动程序是 BSP 设计与开发中另一个非常关键的环节。

### 3) 系统软件层

系统软件层由实时多任务操作系统 (Real-time Operation System, RTOS)、文件系统、图形用户接口 (Graphic User Interface, GUI)、网络系统及通用组件模块组成。RTOS 是嵌入式应用软件的基础和开发平台。

#### (1) 嵌入式操作系统

嵌入式操作系统 (Embedded Operating System, EOS) 是一种用途广泛的系统软件，过去它主要应用于工业控制和国防系统领域。EOS 负责嵌入系统的全部软、硬件资源的分配、任务调度，控制、协调并发活动。它必须体现其所在系统的特征，能够通过装卸某些模块来达到系统所要求的功能。目前，已推出一些应用比较成功的 EOS 产品系列。随着 Internet 技术的发展、信息家电的普及应用及 EOS 的微型化和专业化，EOS 开始从单一的弱功能向高专业化的强功能方向发展。嵌入式操作系统在系统实时高效性、硬件的相关依赖性、软件固化以及应用的专用性等方面具有较为突出的特点。EOS 是相对于一般操作系统而言的，它除具备了一般操作系统最基本的功能，如任务调度、同步机制、中断处理、文件处理等外，还有以下特点：

- 可裁减性。支持开放性和可伸缩性的体系结构。
- 强实时性。EOS 实时性一般较强，可用于各种设备控制中。
- 统一的接口。提供设备统一的驱动接口。
- 操作方便、简单、提供友好的图形 GUI 和图形界面，追求易学易用。
- 提供强大的网络功能，支持 TCP/IP 协议及其他协议，提供 TCP/UDP/IP/PPP 协议支持及统一的 MAC 访问层接口，为各种移动计算设备预留接口。
- 强稳定性，弱交互性。嵌入式系统一旦开始运行就不需要用户过多的干预，这就要负责系统管理的 EOS 具有较强的稳定性。嵌入式操作系统的用户接口一般不提供操作命令，它通过系统的调用命令向用户程序提供服务。
- 固化代码。在嵌入式系统中，嵌入式操作系统和应用软件被固化在嵌入式系统计

算机的 ROM 中。

- 更好的硬件适应性，也就是良好的移植性。

## (2) 文件系统

通用操作系统的文件系统通常具有以下功能：

- 提供用户对文件操作的命令。
- 提供用户共享文件的机制。
- 管理文件的存储介质。
- 提供文件的存取控制机制，保障文件及文件系统的安全性。
- 提供文件及文件系统的备份和恢复功能。
- 提供对文件的加密和解密功能。

嵌入式文件系统比较简单，主要提供文件存储、检索和更新等功能，一般不提供保护和加密等安全机制。它以系统调用和命令方式提供文件的各种操作，主要有：

- 设置、修改对文件和目录的存取权限。
- 提供建立、修改、改变和删除目录等服务。
- 提供创建、打开、读写、关闭和撤销文件等服务。

此外嵌入式文件系统还具有以下特点：

- 兼容性。嵌入式文件系统通常支持几种标准的文件系统，如 FAT32、JFFS2、YAFFS 等。
- 实时文件系统。除支持标准的文件系统外，为提高实时性，有些嵌入式文件系统还支持自定义的实时文件系统，这些文件系统一般采用连续的方式存储文件。
- 可裁减、可配置。根据嵌入式系统的要求选择所需的文件系统，选择所需的存储介质，配置可同时打开的最大文件数等。
- 支持多种存储设备。嵌入式系统的外存形式多样，嵌入式文件系统需方便的挂接不同存储设备的驱动程序，具有灵活的设备管理能力。同时根据不同外部存储器的特点，嵌入式文件系统还需考虑其性能、寿命等因素，发挥不同外存的优势，提高存储设备的可靠性和使用寿命。

## (3) 图形用户接口 (GUI)

GUI 的广泛应用是当今计算机发展的重大成就之一，它极大地方便了非专业用户的使用，人们从此不再需要死记硬背大量的命令，取而代之的是可以通过窗口、菜单、按键等方式来方便地进行操作。而嵌入式 GUI 又与 PC 机上的 GUI 有着明显的不同，嵌入式系统的 GUI 具有下面几个方面的基本要求：轻型、占用资源少、高性能、高可靠性、便于移植、可配置等特点。

嵌入式系统中的图形界面，一般采用下面的几种方法实现：

- 针对特定的图形设备输出接口，自行开发相应的功能函数。
- 购买针对特定嵌入式系统的图形中间软件包。
- 采用源码开放的嵌入式 GUI 系统。
- 使用独立软件开发商提供的嵌入式 GUI 产品。

#### 4) 应用软件层

应用软件层是由基于实时系统开发的应用程序组成,用来实现对被控对象的控制功能。功能层是面向被控对象和用户的,为方便用户操作,往往需要提供一个友好的人机界面。

对于一些复杂的系统,在系统设计的初期阶段就要对系统的需求进行分析,确定系统的功能,然后将系统的功能映射到整个系统的硬件、软件和执行装置的设计过程中,称为系统的功能实现。

### 1.1.4 实时系统

实时系统越来越广泛地应用于各个领域例,也越来越受到人们的重视,包括航空、航天、工业过程控制、武器防御系统、自动化导航/控制系统、医疗、信息检索、银行、多媒体系统等领域。这些实时系统有多种形式,从简化的指令控制到复杂的控制系统,如核电站控制系统等,这些多样化的系统有个共同特点和要求就是系统中的任务不但执行结果要正确,而且它们必须在一定的时间约束(Deadline,即“截止期限”)内完成。

实时系统与通用计算机系统不同的是:通用系统一般追求的是系统的平均响应时间和用户使用系统的方便,而实时系统主要考虑的是系统在最坏情况下的系统行为。一个逻辑上正确的计算结果,若其产生的时间晚于某个规定的时间,那么也认为系统的行为是不正确的。

#### 1. 实时系统定义

实时系统(Real-time operating system, RTOS)的正确性不仅依赖于系统计算的逻辑结果,还依赖于产生这个结果的时间。实时系统能够在指定或者确定的时间内完成系统功能和对外部或内部、同步或异步时间做出响应的系统。因此实时系统应该有在事先定义的时间范围内识别和处理离散事件的能力;系统能够处理和存储控制系统所需要的大量数据。

#### 2. 实时系统特点

##### (1) 时间约束性

实时系统的任务具有一定的时间约束(截止期限)。根据截止期限,实时系统的实时性可分为“硬实时”和“软实时”。硬实时是指应用的时间需求应能够得到完全满足,否则就造成重大安全事故,甚至造成重大的生命财产损失和生态破坏,如在航空航天、军事、核工业等一些关键领域中的应用。软实时是指某些应用虽然提出了时间需求,但实时任

务偶尔违反这种需求对系统运行及环境不会造成严重影响，如监控系统等和信息采集系统等。

### (2) 可预测性

可预测性是指系统能对实时任务的执行时间进行判断，确定是否能够满足任务的时限要求。由于实时系统对时间约束要求的严格性，使可预测性成为实时系统的一项重要性能要求。除了要求硬件延迟的可预测性以外，还要求软件系统的可预测性，包括应用程序的响应时间是可预测的，即在有限的时间内完成必须的工作；以及操作系统的可预测性，即实时原语、调度函数等运行开销应是有界的，以保证应用程序执行时间的有界性。

### (3) 可靠性

大多数实时系统要求有较高的可靠性。在一些重要的实时应用中，任何不可靠因素和计算机的一个微小故障，或某些特定强实时任务（又叫关键任务）超过时限，都可能引起难以预测的严重后果。为此，系统需要采用静态分析和保留资源的方法及冗余配置，使系统在最坏情况下都能正常工作或避免损失。可靠性已成为衡量实时系统性能不可缺少的重要指标。

### (4) 与外部环境的交互作用性

实时系统通常运行在一定的环境下，外部环境是实时系统不可缺少的一个组成部分。计算机子系统一般是控制系统，它必须在规定的时间内对外部请求做出反应。外部物理环境往往是被控子系统，两者相互作用构成完整的实时系统。大多数控制子系统必须连续运转以保证子系统的正常工作或准备对任何异常行为采取动作。

早期的实时系统功能简单，包括单板机、单片机，以及简单的嵌入式实时系统等，其调度过程相对简单。随着实时系统应用范围的不断扩大，系统复杂性不断提高，实时系统具有以下新特点。

#### (1) 多种任务类型

在实时系统中，不但包括周期任务、偶发任务、非周期任务，还包括非实时任务。实时任务要求要满足其时限，而非实时任务要求要使其响应时间尽可能的短。多种类型任务的混合，使系统的可调度性分析更加困难。

#### (2) 约束的复杂性

任务的约束包括时间约束、资源约束、执行顺序约束和性能约束。时间约束是任何实时系统都固有的约束。资源约束是指多个实时任务共享有限的资源时，必须按照一定的资源访问控制协议进行同步，以避免死锁和高优先级任务被低优先级任务堵塞的时间（即优先级倒置时间）不可预测。执行顺序约束是指各任务的启动和执行必须满足一定的时间和顺序约束。例如，在分布式端到端（end-to-end）实时系统中，同一任务的各子任务之间存在前驱/后继约束关系，需要执行同步协议来管理子任务的启动和控制子任务的执行，使它

们满足时间约束和系统可调度性要求。性能约束是指必须满足如可靠性、可用性、可预测性、服务质量（Quality of Service, QoS）等性能指标。

### （3）具有短暂超载的特点

在实时系统中，即使一个功能设计合理、资源充足的系统也可能由于以下原因超载：

- 系统元件出现老化，外围设备错误或系统发生故障。随着系统运行时间的增长，系统元件可能出现老化，系统部件可能发生故障，导致系统可用资源降低，不能满足实时任务的时间约束要求。
- 环境的动态变化。由于不能对未来的环境、系统状态进行正确有效的预测，因此不能从整体角度上对任务进行调度，可能导致系统超载。
- 应用规模的扩大。原先满足实时任务时限要求的系统，随着应用规模的增大，可能出现不能满足任务时限要求的情况，而重新设计、重建系统在时间和经济上又不允许。

## 3. 实时系统调度

为了精确管理“时间”资源，以达到实时性和可预测性要求，并能够满足实时系统的新要求，需用实时调度理论对任务进行调度和可调度性分析。任务调度技术包括调度策略和可调度性分析方法，两者是紧密结合的。任务调度技术研究的范围包括任务使用系统资源（包括处理机、内存、I/O、网络等资源）的策略和机制，以及提供判断系统性能是否可预测的方法和手段。例如，什么时候调度任务运行、在哪运行（当系统为多处理机系统或分布式系统时）、运行多长时间等等；以及判断分析用一定参数描述的实时任务能否被系统正确调度。

给定一组实时任务和系统资源，确定每个任务何时何地执行的整个过程就是调度。在非实时系统中，调度的主要目的是缩短系统平均响应时间，提高系统资源利用率，或优化某一项指标；而实时系统中调度的目的则是要尽可能地保证每个任务满足它们的时间约束，及时对外部请求做出响应。实时调度技术通常有多种划分方法，常用的有以下两种。

### （1）抢占式调度和非抢占式调度

抢占式调度通常是优先级驱动的调度。每个任务都有优先级，任何时候具有最高优先级且已启动的任务先执行。一个正在执行的任务放弃处理器的条件为：自愿放弃处理器（等待资源或执行完毕）；有高优先级任务启动，该高优先级任务将抢占其执行。除了共享资源的临界段之外，高优先级任务一旦准备就绪，可在任何时候抢占低优先级任务的执行。抢占式调度的优点是实时性好、反应快，调度算法相对简单，可优先保证高优先级任务的时间约束，其缺点是上下文切换多。而非抢占式调度是指不允许任务在执行期间被中断，任务一旦占用处理器就必须执行完毕或自愿放弃。其优点是上下文切换少；缺点是在一般情况下，处理器有效资源利用率低，可调度性不好。

## (2) 静态表驱动策略和优先级驱动策略

静态表驱动策略 (Static Table-Driven Scheduling) 是一种离线调度策略, 指在系统运行前根据各任务的时间约束及关联关系, 采用某种搜索策略生成一张运行时刻表。这张运行时刻表与列车的运行时刻表类似, 指明了各任务的起始运行时刻及运行时间。运行时刻表一旦生成就不再发生变化了。在系统运行时, 调度器只需根据这张时刻表启动相应的任务即可。由于所有调度策略在离线情况下制定, 因此调度器的功能被弱化, 只具有分派器 (Dispatcher) 的功能。

优先级驱动策略指按照任务优先级的高低确定任务的执行顺序。优先级驱动策略又分为静态优先级调度策略和动态优先级调度策略。静态优先级调度是指任务的优先级分配好之后, 在任务的运行过程中, 优先级不会发生改变。静态优先级调度又称为固定优先级调度。动态优先级调度是指任务的优先级可以随着时间或系统状态的变化而发生变化。

## 4. 实时系统分类

实时系统主要分为以下两类。

- 强实时 (Hard Real-Time) 系统: 在航空航天、军事、核工业等一些关键领域中, 应用的时间需求应能够得到完全满足, 否则就造成如飞机失事等重大安全事故, 造成重大的生命财产损失和生态破坏。因此, 在这类系统的设计和实现过程中, 应采用各种分析、模拟及形式化验证方法对系统进行严格的检验, 以保证在各种情况下应用的时间需求和功能需求都能够得到满足。
- 弱实时 (Soft Real-Time) 系统: 某些应用虽然提出了时间需求, 但实时任务偶尔违反这种需求对系统的运行以及环境不会造成严重影响, 如视频点播 (Video-On-Demand, VOD) 系统、信息采集与检索系统就是典型的弱实时系统。在 VOD 系统中, 系统只需保证在绝大多数情况下视频数据能够及时传输给用户即可, 偶尔的数据传输延迟对用户不会造成很大影响, 也不会造成像飞机失事一样严重的后果。

## 5. 实时任务分类

在实时系统中, 一个应用通常由一组任务构成, 每个任务完成应用中的一部分功能, 组合后为用户提供特定的服务。实时任务的分类方法有多种。

根据任务的周期划分, 可以分为 3 类。

- 周期任务: 周期任务是指按一定周期到达并请求运行, 每次请求称为任务的一个任务实例, 任务实例所属任务的起始时刻称为该任务实例的到达时刻, 任务实例被置为就绪态的时刻称为该任务实例的释放时刻。
- 偶发任务: 在偶发任务中, 虽然其任务实例的到达时刻不是严格周期的, 但相邻任务实例到达时刻的时间间隔一定大于等于某个最小值, 即偶发任务的各任务实



例按照不高于某个值的速率到达。因此在实际应用中，偶发任务经常被当作周期任务进行处理，其周期为相邻任务实例到达时刻的最小时间间隔。

- 非周期任务：非周期任务是指随机到达系统的任务。

在实时系统中，如果一个任务未能在截止期限前完成，那么称该任务超时。

根据是否允许任务超时，以及超时后对系统造成的影响，任务又分为以下 4 类。

- 强实时任务 (Hard Real-Time Task)：通常是指那些必须在规定的时间内完成的任务，不允许它的任何任务实例超时。若有任务实例未在截止期限内完成，则会对系统造成不可估量的损失。一般采用在最坏情况下任务的响应时间对强实时任务进行可调度性分析。如果存在最大响应时间大于截止期限的任务，则认为该系统不可调度。
- 准实时任务 (Firm Real-Time Task)：通常是指允许任务超时，但若任务超时，则该任务的计算结果没有任何意义。
- 弱实时任务 (Soft Real-Time Task)：通常允许任务超时，但超时后的计算结果仍然有一定的意义，并且其意义随着超时时间的增加而下降。
- 弱-强实时任务 (Weakly Hard Real-Time Task)：弱-强实时任务通常是周期任务，并且具有允许周期任务的一些任务实例超时，但这些超时的任务实例的分布应满足一定的规律的特性。将这种要求称为超时分布约束。若不满足超时分布约束，则会造成系统动态失效。在本文中，任务实例和请求是相互通用的。

实时系统从单用途专用系统向多用途通用操作系统（如实时 Linux 等）发展。实时系统从只支持强实时及其应用发展到既支持强实时也支持弱实时及其应用方面，如开放实时系统的服务质量 (QoS)、多媒体应用、复杂分布式实时系统等。现在使用的实时系统包括实时内核 ( $\mu\text{C}/\text{OS}$  等)、组合的内核、通用操作系统的实时变种（如 RT-Linux, RTAI-Linux, 实时 Windows-NT、XP）等。目前很多实时系统遵循 Posix 实时扩展的工业标准，如 RT-Linux 等。

实时系统一般分为：小而快速的专用内核，如  $\mu\text{C}/\text{OS}$ ；通用操作系统的实时扩展，如 RTLinux 等；基于组件的内核，如 OS-Kit、Coyote、2K、MMLite 等；基于 QoS 的内核等。

实时系统一般是反应式系统，即它们不断地接受外部的输入，并对输入作出反应，它们的任务是维持系统和环境之间不断进行的交互过程。实时系统也可以被视为事件/响应系统。实时系统的行为由一个事件序列、相应的响应及响应的时间限制来定义。

实时系统的反应模型体现出它的通用结构模型，即一般由三类任务组成：数据采集管理任务、数据处理任务、执行机构管理任务。一般的流程是数据采集任务实现传感器数据的采集，数据处理任务处理采集的数据，并把加工后的数据送到执行机构执行。

## 1.2 嵌入式微处理器体系结构

### 1.2.1 冯·诺依曼与哈佛结构

#### 1. 冯·诺依曼结构

传统计算机采用冯·诺依曼 (Von Neumann) 结构, 也称普林斯顿结构, 是一种将程序指令存储器和数据存储器合并在一起的存储器结构。冯·诺依曼结构的计算机其程序和数据共用一个存储空间, 程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置; 采用单一的地址及数据总线, 程序指令和数据的宽度相同。处理器执行指令时, 先从存储器中取出指令解码, 再取操作数执行运算, 即使单条指令也要耗费几个甚至几十个周期, 在高速运算时, 在传输通道上会出现瓶颈效应。

如图 1-3 所示, 冯·诺依曼结构的计算机由 CPU 和存储器构成, 程序计数器 (PC) 是 CPU 内部指示指令和数据存储位置的寄存器。CPU 通过程序计数器提供的地址信息, 对存储器进行寻址, 找到所需要的指令或数据, 然后对指令进行译码, 最后执行指令规定的操作。

在这种体系结构中, 程序计数器只负责提供程序执行所需要的指令或数据, 而不决定程序流程。要控制程序流程, 则必须修改指令。

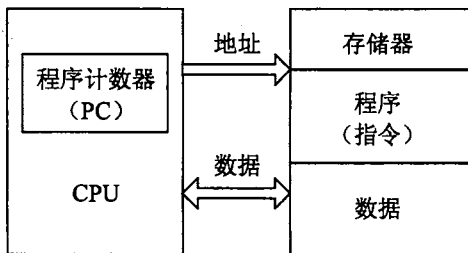


图 1-3 冯·诺依曼体系结构

目前使用冯·诺依曼结构的 CPU 和微控制器有很多。其中包括英特尔公司的 8086 及其他 CPU, ARM 公司的 ARM7、MIPS 公司的 MIPS 处理器也采用了冯·诺依曼结构。

#### 2. 哈佛结构

哈佛 (Harvard) 结构是一种将程序指令存储和数据存储分开的存储器结构。哈佛结构是一种并行体系结构, 它的主要特点是将程序和数据存储在不同的存储空间中, 即程序存储器和数据存储器是两个相互独立的存储器, 每个存储器独立编址、独立访问。与两个存储器相对应的是系统中的 4 套总线: 程序的数据总线与地址总线, 数据的数据总线与地址总线。这种分离的程序总线和数据总线可允许在一个机器周期内同时获取指令字 (来自程序存储器) 和操作数 (来自数据存储器), 从而提高了执行速度, 使数据的吞吐率提高了 1 倍。又由于程序和数据存储器在两个分开的物理空间中, 因此取指和执行能完全重叠。

如图 1-4 所示, 哈佛结构的计算机由 CPU、程序存储器和数据存储器组成, 程序存储器和数据存储器采用不同的总线, 从而提供了较大的存储器带宽, 使数据的移动和交

换更加方便, 尤其提供了较高的数字信号处理性能。

哈佛结构的 CPU 通常具有较高的执行效率。目前使用哈佛结构的 CPU 和微控制器有很多, 除了所有的 DSP 处理器, 还有摩托罗拉公司的 MC68 系列、Zilog 公司的 Z8 系列、ATMEL 公司的 AVR 系列和 ARM 公司的 ARM9、ARM10 和 ARM11 等。

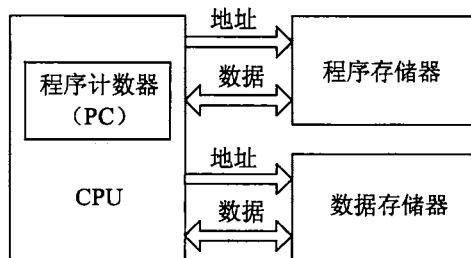


图 1-4 哈佛体系结构

## 1.2.2 CISC 与 RISC

### 1. 复杂指令集计算机 (Complex Instruction Set Computer, CISC)

早期的计算机部件非常昂贵, 主频低, 运算速度慢。为了提高运算速度, 人们不得不再将越来越多的复杂指令加入到指令系统中, 以提高计算机的处理效率, 这就逐渐形成了复杂指令集计算机体系。为了在有限的指令长度内实现更多的指令, 人们又设计了操作码扩展。然后, 为了达到操作码扩展的先决条件——减少地址码, 设计师又发明了各种寻址方式, 如基址寻址、相对寻址等, 以最大限度地压缩地址码长度, 为操作码留出空间。Intel 公司的 X86 系列 CPU 是典型的 CISC 体系的结构, 从最初的 8086 到后来的 Pentium 系列, 每出一代新的 CPU, 都会有自己新的指令, 而为了兼容以前 CPU 平台上的软件, 旧的 CPU 的指令集又必须保留, 这就使指令的解码系统越来越复杂。CISC 可以有效地减少编译代码中指令的数目, 使取指令操作所需要的内存访问数量达到最小化。此外, CISC 可以简化编译器结构, 它在处理器指令集中包含了类似于程序设计语言结构的复杂指令, 这些复杂指令减少了程序设计语言和机器语言之间的语义差别, 而且简化了编译器的结构。

为了支持复杂指令集, CISC 通常包括一个复杂的数据通路和一个微程序控制器。如图 1-5 所示。微程序控制器由一个微程序存储器、一个微程序计数器 (MicroPC) 和地址选择逻辑构成。在微程序存储器中的每一个字都表示一个控制字, 并且包含了一个时钟周期内所有数据通路控制信号的值。这就意味着控制字中的每一位表示一个数据通路控制线的值。例如, 它可以用于加载寄存器或是选择 ALU 中的一个操作。此外, 每个处理器指令都由一系列的控制字组成。当从内存中取出这样的一条指令时, 首先把它放在指令寄存器中, 然后地址选择逻辑再根据它来确定微程序存储器中相应的控制字顺序起始地址。当把该起始地址放入 MicroPC 中后, 就从微程序内存中找到相应的控制字, 并利用它在数据通路中把数据从一个寄存器传送到另一个寄存器。由于 MicroPC 中的地址并发递增来指向下一个控制字, 因此对于序列中的每个控制器都会重复一遍这一步骤。最终, 当执行完最后一个控制字时, 就从内存中取出一条新的指令, 整个过程会重复进行。

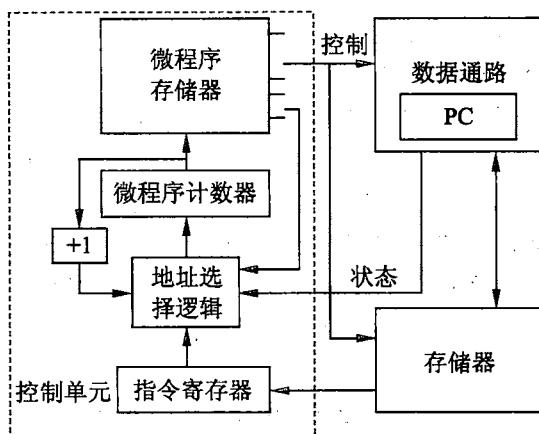


图 1-5 微程序控制的 CISC 计算机

由此可见，控制字的数量及时钟周期的数目对于每一条指令都可以是不同的。因此，在 CISC 中很难实现指令流水操作。另外，速度相对较慢的微程序存储器需要一个较长的时钟周期。由于指令流水和短的时钟周期都是快速执行程序的必要条件，因此 CISC 体系结构对于高效处理器而言是不太合适的。

CISC 体系结构的计算机存在以下一些问题。

#### (1) 指令的 2/8 规律

CISC 计算机中，各种指令的使用频率相差悬殊。大量的统计数字表明，大概有 20% 的比较简单指令被反复使用，使用量约占整个程序的 80%；而有 80% 左右的指令则很少使用，其使用量约占整个程序的 20%。

#### (2) VLSI 制造工艺要求 CPU 控制逻辑的规整性

进入 20 世纪 80 年代后，VLSI 技术的发展非常迅速，往往每 3 到 4 年集成度就提高一个数量级。VLSI 工艺要求规整性，而 CISC 处理器中，为了实现大量复杂的指令，控制逻辑极不规整，给 VLSI 工艺造成很大困难。

此外，以 CISC 处理器中，大量使用微程序技术以实现复杂的指令系统。20 世纪 70 年代之前，一般采用磁芯做主存储器，采用半导体做控制存储器，两者的速度相差 5~10 倍。从 70 年代后期开始，大量使用 DRAM（动态随机存储器）做主存储器，使主存与控制存储器的速度相当，从而使许多简单指令没有必要用微程序来实现。而复杂的指令，用微程序实现和用简单指令组成的子程序实现已经没有什么区别。

#### (3) 软硬件的功能划分

在 CISC 中，为了支持目标程序的优化，支持高级语言和编译程序，增加了许多复杂的指令，用一条指令来代替一串指令。这些复杂指令简化目标程序，缩小了高级语言与机



器指令之间的语义差距。但是,增加了这些复杂指令并不等于有利于缩短程序的执行时间。

为了实现复杂的指令,不仅增加了硬件的复杂程序,而且使指令的执行周期大大加长。例如,为了支持编译程序的对称性要求,一般的运算型指令都能直接访问主存储器,从而使指令的执行周期数增加,数据的重复利用率降低。据统计,一般 CISC 处理器的指令平均执行周期都在 4 以上,有些在 10 以上,如 Intel 公司的 8088 等。

这里有一个软件与硬件的功能如何恰当分配的问题。在 CISC 中,通过增强指令系统的功能,简化了软件,增加了硬件的复杂程度。然而,由于指令复杂了,指令的执行时间必然加长,从而有可能使整个程序的执行时间反而增加,因此,在计算机体系结构设计中,软硬件功能划分必须合适。

## 2. 精简指令集计算机 (Reduced Instruction Set Computer, RISC)

### (1) RISC 的产生

1975 年 IBM 公司开始研究指令系统的合理性问题,IBM 公司的 John cocke 提出精简指令系统的想法,并于 1979 年研制出一种用于电话交换系统的 32 位小型计算机 IBM801,它有 120 条指令,工作速度为 10MIPS,这是世界是第一台采用 RISC 思想的计算机。1979 年,美国加州伯克利大学的 David Patterson 开展了 RISC 的研究工作,并研制了 RISC I 和 RISC II 机,后来斯坦福大学成功研制了 MIPS 机,这些都为 RISC 的诞生与发展起了很大作用。

### (2) RISC 的发展

1983 年以后,一些中、小型公司开始推出 RISC 产品,由于它具有高性价比,市场占有率不断提高。1987 年 SUN 微系统公司用 SPARC 芯片构成工作站,从而使其工作站的销售量居于世界首位。当前一些大公司,如 IBM、DEC、Intel 和 Motorola 等都将其部分力量转到 RISC 方面来,RISC 已经成为当前计算机发展不可逆转的趋势。一些发展较早的大公司转到 RISC 上来还需要考虑其他因素,如因为 RISC 与 CISC 指令系统不兼容,因此他们在 CISC 上开发的大量软件如何转到 RISC 平台上来是首先要考虑的;而且这些公司的操作系统专用性很强,又比较复杂,这给软件的移植带来了麻烦。而 SUN 微系统公司,因为其以 UNIX 操作系统作为基础,软件移植比较容易,因此其工作站的重点很快从 CISC 转移到 RISC。

从技术发展的角度,CISC 技术已很难再有突破性的大进展,要想大幅度提高性价比也很困难。而 RISC 技术是在 CISC 基础上发展起来的,且发展势头正猛。在 CISC 市场上占有率最高的 Intel 公司和 Motorola 公司也进军 RISC 领域。IBM 公司、Motorola 公司和 Apple 公司联合发展 Power PC 芯片,HP 公司和 Intel 公司联合开发代号为 Merced 的微处理器。

### (3) RISC 的特点

RISC 的着眼点不是简单地放在简化指令系统上,而是通过简化指令系统使计算机的结

构更加简单合理,从而提高运算效率。

如图 1-6 所示, RISC 处理器的数据通路通常由一个大的寄存器文件和一个 ALU 组成, 一个大的寄存器文件是十分必要的, 因为它包含了程序计算中所有的操作数和结果。通过 Load 指令将数据放到寄存器文件中, 通过 Store 指令将其放回内存。寄存器文件越大, 代码中的 Load 和 Store 指令的数目就越少。当 RISC 执行一序列指令时, 指令管道首先将指令放到指令寄存器中, 然后将该指令解码并从寄存器文件中操作数, 最后, RISC 做下面两件事情之一: 或者在 ALU 中执行所需的操作, 或者从数据缓存里面读/写数据。应注意每个指令的执行仅仅占用大约 3 个时钟周期, 这就意味着指令流水线可能短小且高效。而且仅在数据分支相关性的情况下才会使用较多的时钟周期。

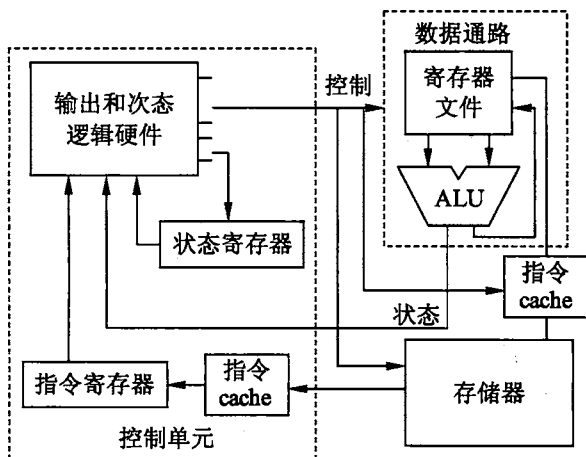


图 1-6 硬件控制的 RISC 计算机

同时由于所有的操作数都包含在寄存器文件中, 而且只使用了几种简单的寻址方式, 所以同样可以简化数据通路的设计。此外, 因为每个操作要执行一个时钟周期而每个指令要执行 3 个时钟周期, 因此控制单元也可以很简单, 而且可能使用随机逻辑而不是微程序控制来实现。总之, RISC 中控制和数据通路的简化导致了简短的时钟周期, 并最终达到了更高的性能。

然而, RISC 体系结构的简化则要求一个更为复杂的编译器。例如, RISC 设计不会在指令相关性发生时就停止指令流水线, 这就意味着编译器有责任产生出无相关性的代码, 或者可以通过时延指令的产生, 或者对指令进行重新排序。而且由于指令数目的减少, RISC 编译器将需要一系列的 RISC 指令来完成复杂的操作, 也给编译器带来了灵活的优化性能。

计算机执行程序所需的时间  $P$  可以用下式计算:



$$P = I \times CPI \times T$$

其中  $I$  是高级语言程序编译后在机器上运行的指令数,  $CPI$  为执行每条指令所需的平均周期数,  $T$  是每个机器周期的时间。

由于 RISC 指令比较简单, 原 CISC 中比较复杂的指令在这里用子程序来代替, 因此 CISC 的  $I$  要比 CISC 多 20%~40%, 但是 RISC 的大多数指令只用一个机器周期实现, 所以  $CPI$  的值要比 CISC 小的多。同时因为 CISC 结构简单, 所以完成一个操作所经过的数据通路较短, 使得  $T$  的值大为减少。后来 RISC 的硬件结构有很大改进, 一个机器周期平均可完成 1 条以上指令, 甚至可以达到几条指令。

RISC 是在继承 CISC 的成功技术并克服 CISC 的缺点的基础上产生并发展起来的, 大部分 RISC 具有以下特点:

- 优先选取使用频率最高的一些指令, 以及一些很有用但不复杂的指令, 避免使用复杂指令。
- 指令长度固定, 指令格式种类少, 寻址方式种类少。指令之间各字段的划分比较一致, 各字段的功能也比较规整。
- 只有 Load/Store 指令能够访问存储器, 其余指令的操作都在寄存器之间进行。
- CPU 中通用寄存器数量相当多, 算术逻辑运算指令的操作数都在通用寄存器中存取。
- 大部分指令在一个或小于一个机器周期内完成。
- 以硬布线控制逻辑为主, 不用或少用微码控制。
- 一般用高级语言编程, 特别重视编译优化工作, 以减少程序执行时间。

表 1-1 列出了 CISC 与 RISC 计算机的特点。

表 1-1 CISC 与 RISC 的特点

类别	CISC	RISC
指令系统	指令数量很多	较少, 通常少于 100
执行时间	有些指令执行时间很长, 如整块的存储器内容复制; 或将多个寄存器的内容复制到存储器	没有较长执行时间的指令
编码长度	编码长度可变, 1~15 字节	编码长度固定, 通常为 4 个字节
寻址方式	寻址方式多样	简单寻址
操作	可以对存储器和寄存器进行算术和逻辑操作	只能对寄存器进行算术和逻辑操作, Load/Store 体系结构
编译	难以用优化编译器生成高效的目标代码程序	采用优化编译技术, 生成高效的目标代码程序

当然,和 CISC 架构相比较,尽管 RISC 架构有上述的优点,但决不能认为 RISC 架构就可以取代 CISC 架构。事实上, RISC 和 CISC 各有优势,而且界限并不那么明显。现代的 CPU 往往采用 CISC 的外围,内部加入了 RISC 的特性,如超长指令集 CPU 就是融合了 RISC 和 CISC 的优势,成为未来的 CPU 发展方向之一。

在 PC 机和服务器领域,以 X86 为代表的 CISC 体系结构是市场的主流。在嵌入式系统领域,由于降低成本和功耗比保持向下兼容更为重要, RISC 结构的微处理器将占有重要的位置。

### 1.2.3 流水线技术

指令流水线就是将一条指令分解成一连串执行的子过程。在 CPU 中把一条指令的串行执行子过程变为若干条指令的子过程在 CPU 中重叠执行,这就是指令流水线的思想。如果能做到每条指令均分解为  $m$  个子过程,且每个子过程的执行时间都一样,则利用此条流水线可将一条指令的执行时间由原来的  $T$  缩短为  $T/m$ 。

#### 1. 流水线的基本概念

流水线技术是将一个重复的时序分解成若干个子过程,而每一个子过程都可有效地在其专用功能段上与其他子过程同时执行。流水线技术应用于计算机系统结构的各个方面,在此以指令的执行过程为例介绍流水线技术。

前面已介绍过,可将指令的执行过程分解成取指令、分析指令和执行指令 3 个子过程。早期指令的执行是顺序方式,即现行指令执行完毕后才开始读取后续指令,这种处理方式控制简单,且比较直观,但在时间安排上不能充分利用各部件。为了提高工作速度,现在的数字计算机都在不同程度上采取重叠处理方式,重叠的程度取决于存储体与运算部件的多少及控制指令部件的工作方式。

在一次重叠处理时,可将指令的执行过程粗分为分析和执行两个子过程,当第  $I$  条指令在指令部件中分析完毕后,将进入执行部件去实现指令的操作。此时指令分析部件处理“空闲”状态,若利用这个“空闲”状态对第  $I+1$  条指令进行分析,使其与第  $I$  条指令的执行同步进行,也即两条指令在时间上存在着重叠,如图 1-7 所示。



图 1-7 一次重叠处理

若把指令的执行过程进一步细分为取指令、指令译码、取操作数和执行 4 个子过程,并改进运算器的结构以加快其执行子过程,则得到如图 1-8 所示的流水处理的时空图,其中的 1、2、3、4、5 表示要处理的 5 条指令。一次重叠与流水的区别在于前者将一条指令的执行分为两个子过程,而后者却分为 4 个或多个子过程。一次重叠可同时执行两条指令,而流水方式则可同时执行多条指令。

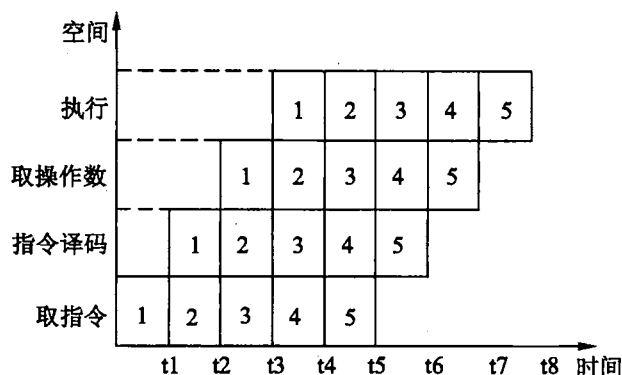


图 1-8 流水线处理的时空图

## 2. 流水线技术的特点

- 流水线可分成若干相互联系的子过程。
- 实现子过程的功能所需时间尽可能相等。
- 形成流水处理，需要一段准备时间。
- 指令流发生不能顺序执行时，会使流水线过程中断，再形成流水线过程则需要时间。

## 3. 流水线结构的分类

流水线结构的类型众多，并且分类方法各异，常见的几种分类方法如下所述。

### (1) 按完成的功能分类

- 单功能流水线：只完成一种固定功能的流水线，如只能实现浮点加。
- 多功能流水线：同一流水线上可有多种连接方式来实现多种功能，如 TI 公司的 ASC 运算器中的 8 个可并行工作的功能块，可按不同的连接方式实现浮点加、减或定点乘法运算。

### (2) 按同一时间内各段之间的连接方式分类

- 静态流水线：同一时间流水线上的所有功能块只能按同一种运算的连接方式工作。如 ASC 运算器中的 8 个可并行工作的功能块，或都按浮点加、减运算连接，或都按定点乘法运算连接。
- 动态流水线：同一时间流水线上的所有功能块可按不同种运算的连接方式工作。

### (3) 按数据表示分类

- 标量流水线处理器：只能对标量数据进行流水处理。
- 向量流水线处理器：它具有向量指令，可对向量的各元素进行流水处理。

#### 4. 流水线处理机的主要指标

##### (1) 吞吐率

吞吐率是指单位时间里流水线处理机流出的结果数。对指令而言就是单位时间里执行的指令数。如果流水线的子过程所用时间不一样长, 则吞吐率  $P$  应为最长子过程的倒数, 即:

$$P = 1/\max\{\Delta t_1, \Delta t_2, \dots, \Delta t_m\}$$

##### (2) 建立时间

流水线开始工作, 须经过一定时间才能达到最大吞吐率, 这就是建立时间。若  $m$  个子过程所用时间一样, 均为  $t_0$ , 则建立时间  $T_0 = m\Delta t_0$ 。

### 1.2.4 信息存储的字节顺序

#### 1. 大端和小端存储法

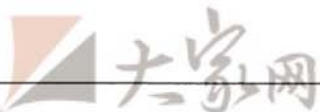
大多数计算机使用称为字节的 8 位 (bit) 的数据块做为最小的可寻址的存储器单位, 而不是访问存储器中单独的位。存储器的每一个字节都由一个唯一的数字来标识, 称为它的地址 (address), 所有可能地址的集合称为存储器空间。对于软件而言, 它将存储器看作一个大的字节数组, 称为虚拟存储器。在实际应用中, 虚拟存储器可以划分成不同的单元, 来存放程序、指令和数据等信息。这些信息的位置完全是由地址决定的。如 C 语言中一个指针的值, 就是它所指向的某个存储块第一个字节的虚拟地址。

每一种微处理器都用一个字长 (word) 表明整数和指令数据的大小。字长决定了微处理器的寻址能力, 即虚拟地址空间的大小。对于一个字长为  $n$  位的微处理器, 它的虚拟地址范围为  $0 \sim 2^n - 1$ 。也就是说, 32 位的微处理器, 其可访问的虚拟地址空间为  $2^{32}$ , 即 4GB。

微处理器和编译器使用不同的方式来编码数据, 如不同长度的整数和浮点数, 从而支持多种数据格式。以 C 语言为例, 它支持整数和浮点数等多种数据格式。int 在 C 中表示整数类型数据, 在 int 之前还可以加上 long 和 short 等限定词, 以表示各种整数和大小, 表 1-2 给出了 C 中常用的数据类型表示。

表 1-2 C 语言中常用数据类型及其占内存情况

C 的数据类型声明	通常在 32 位微处理器中所占的字节数	C 的数据类型声明	通常在 32 位微处理器中所占的字节数
char	1	long int	4
short int	2	float	4
int	4	double	8



对于多于一个字节类型的数据，在存储器中有两种存放方法。一种是低字节数据存放在内存低地址处，高字节数据存放在内存高地址处，称为小端字节顺序存储法；另一种是高字节数据存放在低地址处，低字节数据存放在高地址处，称为大端字节顺序存储法。这有点像我们写字，有人习惯从左向右写，有人习惯从右向左写。

例如，假设一个 32 位字长的微处理器上定义一个 int 类型的常量 a，其内存地址位于 0x8000 处，其值用十六进制表示为 0x01234567。如图 1-9（a）所示，如果按小端法存储，则其最低字节数据 0x67 存放在内存低地址 0x8000 处，最高字节数据 0x01 存放在内存高地址 0x8003 处。如图 1-9（b）所示，如果按大端法存储，则其最高字节数据 0x01 存放在内存的低地址 0x8000 处，而最低字节数据 0x67 存放在内存的高地址 0x8003 处。

地址	0x8000	0x8001	0x8002	0x8003
数据（十六进制表示）	0x67	0x45	0x23	0x01
数据（二进制表示）	01100111	01000101	00100011	00000001

（a）小端存储法

地址	0x8000	0x8001	0x8002	0x8003
数据（十六进制表示）	0x01	0x23	0x45	0x67
数据（二进制表示）	00000001	00100011	01000101	01100111

（b）大端存储法

图 1-9 大端和小端存储法示例

显然，选择大端存储法还是小端存储法，并不存在技术原因，只是涉及到处理器厂商的立场和习惯。Intel 公司 X86 平台的微处理器都采用小端存储法；而 IBM、Motorola 和 Sun Microsystems 公司的大多数微处理器采用大端存储法。IBM 公司采用 Intel 公司处理器制造的计算机，则采用小端法。此外，还有一些微处理器，如 ARM、MIPS 和 Motorola 的 PowerPC 等，可以通过芯片上电启动时确定的字节顺序规则，来选择存储模式。

对于大多数程序员而言，机器的字节存储顺序是完全不可见的，无论哪一种存储模式的微处理器编译出的程序都会得到相同的结果。不过，有些情况下，字节顺序会成为问题。当不同存储模式的微处理器之间通过网络传送二进制数据时，会出现所谓的“UNIX”问题。字符“UNIX”在 16 位字长的微处理器上被表示为两个字节，当被传送到不同存储模式的机器上时，则会变为“NUXI”，这个问题最早是 UNIX 操作系统的早期版本从 PDP-11 到移植到 IBM 机器上时发生的。为了避免这类问题，网络应用程序代码编写必须遵循已建立好的关于字节顺序的规则，以保证发送方微处理器先在其内部将发送的数据转换成网络标准，而接收方微处理器再将网络标准转换为它的内部表示。此外，在使用反汇编器阅读机

器级二进制代码时，在不同存储模式的微处理器上会得到不同的结果。

## 2. 可移植性问题

当在不同存储顺序的微处理器间进行程序移植时，要特别注意存储模式的影响。例如在解释以二进制格式存储的数据和使用合适的掩码时，存储顺序就非常重要，因为使用不同的存储顺序会得出不同的结果。

把从软件得到的二进制数据写成一般的数据格式往往会涉及到存储顺序的问题。例如，存储 BMP 位图文件需要小端法整数，如果数据采用大端法整数存储，那么由于格式不匹配，数据将被破坏。

在多台不同存储顺序的主机之间共享信息可以有两种方式：一种是以单一存储方式共享数据，一种是允许主机以不同的存储方式共享数据，但需要标记它们使用的方式。两种方法都有其优点，一方面，使用单一存储顺序可使解码简单，因为它只要解释一种格式。另一方面，多种存储方式使得编码容易，因为不需要对数据的原顺序进行转化，同时当编码器和解码器采用同一种存储方式时也能提高通信效率，因为不需要变换字节顺序。

## 3. 通信中的存储顺序问题

在网络通信中，Internet 协议（即 IP 协议）定义了标准的网络字节顺序。该字节顺序被用于所有设计使用在 IP 协议上的数据包、高级协议和文件格式上。

Berkeley 应用程序接口定义了一套将 16 位或 32 位整型数据转化成网络字节顺序的函数：函数 `htonl` 和 `htons` 分别将 32 位（长整型）和 16 位（短整型）数值从主机字节顺序转化成网络字节顺序；而函数 `ntohl` 和 `ntohs` 则是将网络字节顺序转化成主机字节顺序。

很多网络设备也存在存储顺序问题：即字节中的位采用大端法（最重要的位优先）或小端法（最不重要的位优先）发送。这取决于 OSI 模型最底层的数据链路层。

## 4. 数据格式的存储顺序

一个典型的例子就是不同的国家采用不同的日期表示方法，美国和其他一些国家，日期格式顺序一般是：月一日一年（如：5 月 24 日 2006 年 或 5/24/2006），这是中间表示法。

在世界大部分国家中，包括除瑞典、拉脱维亚和匈牙利之外的欧洲，日期格式为：日一月一年（比如 24 日 5 月 2006 或 5/24/2006），这是小端表示法。

中国、日本和 ISO 8601 国际正式标准顺序的日期顺序排列顺序是：年一月一日（比如 2006 年 5 月 24 日 或 2006-05-24），这是大端表示法。

ISO 8601 的顺序架构将数据通过计算机进行处理。也就是说，运算排序在处理日期字符串的数字部分和字符串的非数字部分没有区别，并且日期会按年代顺序自动排列。注意：为使其能正常工作，年份必须用 4 位数字表示，月份和日数分别用两位表示。因此，个位数的日和月必须在前面填补一个零，如 01, 02, ..., 09 等。



## 1.3 嵌入式系统的硬件基础

现代计算机内部的电子元件都是数字式的。数字式的电子元件工作状态是二值电平：高电平和低电平。这也是计算机采用二进制的主要原因。通常不指定具体的电平值，而是采用信号来表示，如，用“逻辑真”、“1”或“确定”来表示高电平，而用“逻辑假”、“0”或“不确定”来表示低电平。1 和 0 称为互补信号。

根据电路是否具有存储功能，将逻辑电路划分为两种类型：组合逻辑电路和时序逻辑电路。组合逻辑电路不含存储功能，它的输出值仅取决于当前的输入值；时序逻辑电路含存储功能，它的输出值不仅取决于当前输入状态，还取决于存储单元中的值。

### 1.3.1 组合逻辑电路基础

所谓组合逻辑电路，是指该电路在任一时刻的输出，仅取决于该时刻的输入信号，而与输入信号作用前电路的状态无关。组合逻辑电路一般由门电路组成，不含记忆元件，输入与输出之间无反馈。常用的组合逻辑电路有译码器和多路选择器等。

#### 1. 真值表

由于组合电路中不包含任何存储单元，所以组合电路的输出值可由当前输入值完全确定。这种确定的对应关系可以由真值表（true table）来描述。例如，对于有  $n$  个输入的逻辑电路，对应的真值表有  $2^n$  种输入组合，每一种输入组合表示一组输入状态集，分别对应一个确定的输出。

假设某个逻辑函数，其输入为 A、B、C 3 个端口，输出为 D、E、F 3 个端口。实现的功能为：当 A、B、C 中至少有一位为“1”时，D 端口输出为“1”；当 A、B、C 中至少有两位为“1”时，E 端口输出为“1”；当输入值都为“1”时，F 端口输出为“1”。则该真值表的描述如表 1-3 所示。

表 1-3 一个真值表描述

输 入			输 出			输 入			输 出		
A	B	C	D	E	F	1	0	0	1	0	0
0	0	0	0	0	0	1	0	1	1	1	0
0	0	1	1	0	0	1	1	0	1	1	0
0	1	0	1	0	0	1	1	1	1	1	1
0	1	1	1	1	0						

通常，真值表能够完全描述任何一种组合逻辑函数，但是表的大小随着输入个数的增加呈指数增长，而且不够清晰。

## 2. 布尔代数

描述逻辑函数的另外一种方法是逻辑表达式，可以通过布尔代数（Boolean algebra）实现。布尔代数中有3种典型的操作符：OR、AND 和 NOT。

- OR（“或”）操作符，记为“+”，也称为逻辑和（logical sum）。如  $A+B$ ，若  $A$  和  $B$  中至少有一位为 1 时，则结果为 1。
- AND（“与”）操作符，记为“ $\cdot$ ”，也称为逻辑乘（logical product）。如  $A \cdot B$ ，当且仅当输入值都为 1 时，其结果才为 1。
- NOT（“非”）操作符，记为“ $\bar{A}$ ”，也称为逻辑非。当输入  $A$  为 0 时，输出为 1；当输入为 1 时，输出为 0。

常用的布尔代数定律如表 1-4 所示。

表 1-4 常用的布尔代数定律（基本公式）

表 达 式	名 称	运 算 规 律
$A+0=A$	0-1 律	变量与常量的关系
$A \cdot 0=0$		
$A+1=1$		
$A \cdot 1=A$		
$A+A=A$	同一律	逻辑代数的特殊规律，不同于普通代数
$A \cdot A=A$		
$A+\overline{A}=1$	互补律	
$A \cdot \overline{A}=0$		
$\overline{\overline{A}}=A$	非非律	
$A+B=B+A$	交换律	与普通代数规律相同
$A \cdot B=B \cdot A$		
$(A+B)+C=A+(B+C)$	结合律	
$(A \cdot B) \cdot C=A \cdot (B \cdot C)$		
$A \cdot (B+C)=A \cdot B+A \cdot C$	分配律	
$A+BC=(A+B)(A+C)$		
$\overline{A+B}=\overline{A} \cdot \overline{B}$	反演律（摩根定律）	逻辑代数的特殊规律，不同于普通代数
$\overline{A \cdot B}=\overline{A}+\overline{B}$		

## 3. 门电路

门电路可以实现基本的逻辑功能。基本的门电路如图 1-10 所示，包括与门、或门和非门。

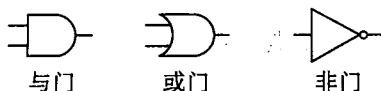


图 1-10 基本门电路

通常在信号的输入或输出端加上一个“ $\circ$ ”表示对输入/输出信号取非。图 1-11 所示都表示逻辑表示式  $\overline{A+B}$ ，右图是左图的简化表示。

任何一个逻辑表达式都可以用与门、非门和或门的组合来表示。如果允许某个门电路取非，那么任何一个逻辑函数都可以仅用与门或仅用或门实现。常见的两种反向门

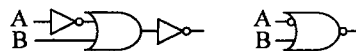


图 1-11 门电路的简化表示

电路为 NOR 和 NAND，它们分别对应或门、与门的取非。NOR 和 NAND 的门电路称为全能门电路，因为任何一种逻辑函数可以用这种门电路得以实现。

#### 4. 译码器

译码器又称为解码器 (decoder)，译码器是一种多输入多输出的组合逻辑网络，它有  $n$  个输入端， $m$  个输出端。与译码器对应的是编码器 (encoder)，它实现的是译码器的逆功能。译码器的框图如图 1-12 所示。

每输入一个  $n$  位的二进制代码，在  $m$  个输出端中最多有一个有效。译码器的输入端和输出端之间应满足下列关系：

$$m \leq 2^n$$

$m=2^n$  时，称为全译码；当  $m<2^n$  时，称为部分译码。

根据逻辑功能的不同，译码器可分为通用译码器和显示译码器两大类。常见的译码器有：

##### (1) 二进制译码器

二进制译码器是一种全译码器，常见的有 2-4 译码器、3-8 译码器和 4-16 译码器等。以 3-8 译码器为例进行介绍。如图 1-13 所示，3-8 译码器有 3 个输入端  $S_1$ 、 $S_2$  和  $S_3$ ，有 8 个输出端  $D_1 \sim D_8$ 。当输入为某一组合时，对应的输出端仅有一个为“1”（或为“0”），其余的输出端均为“0”（或为“1”）。3-8 译码器的真值表如表 1-5 所示。



图 1-12 译码器

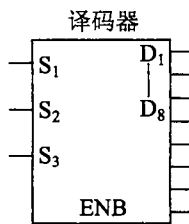


图 1-13 译码器

##### (2) 二-十进制译码器

二-十进制代码译成对应的十进制数码 0~9，称为二-十进制译码器，其  $n=4$ ， $m=10$ ，故属于部分译码。集成的二-十进制译码器芯片有几种，分别有 8421 码输入、余 3 码输入等。

表 1-5 3-8 译码器的真值表

输 入			输 出							
S <sub>1</sub>	S <sub>2</sub>	S <sub>3</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	D <sub>8</sub>
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

### (3) 显示译码器

在数字系统中,经常要用到字符显示器。目前,常用字符显示器有发光二极管 LED 字符显示器和液晶 LCD 字符显示器。

发光二极管是用砷化镓、磷化镓等材料制造的特殊二极管。在发光二极管正向导通时,电子和空穴大量复合,把多余能量以光子形式释放出来,根据材料不同发出不同波长的光。发光二极管既可以用高电平点亮,也可以用低电平驱动,分别如图 1-14 (a) 和 (b) 所示。其中限流电阻一般几百到几千欧姆,由发光亮度(电流)决定。

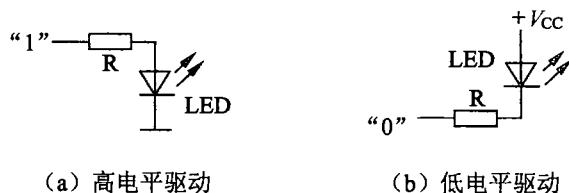


图 1-14 发光二极管驱动电路

将 7 个发光二极管封装在一起,每个发光二极管做成字符的一个段,就是所谓的七段 LED 字符显示器。根据内部连接的不同,LED 显示器有共阴和共阳之分。图 1-15 所示的共阴 LED 显示器适用于高电平驱动,共阳 LED 显示器适用于低电平驱动。由于集成电路的高电平输出电流小,而低电平输出电流相对比较大,采用集成门电路直接驱动 LED 时,较多采用低电平驱动方式。

液晶七段字符显示器 LCD 利用液晶有外加电场和无外加电场时不同的光学特性来显示字符。无外加电场时,液晶排列整齐,入射光大部分反射回来,液晶呈透明状态。外加电场时,液晶因电离而打破分子规则排列,入射光散射,仅一小部分反射回来,液晶呈混

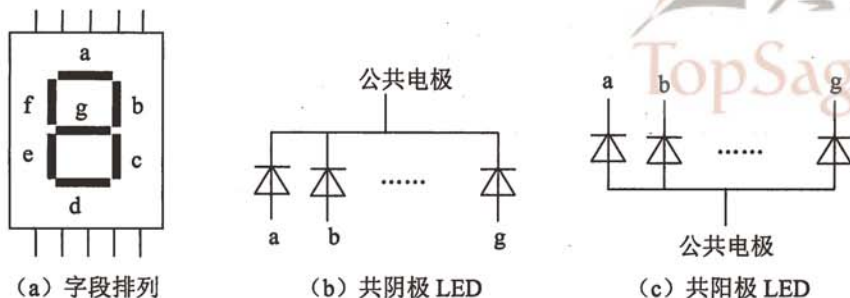


图 1-15 七段字符显示器

浊状态，显示暗灰色。LCD 字符显示器的七段透明电极做成如图 1-15 (a) 形状，并有一公共电极称为背电极也做成此形状。透明电极和背电极之间加电场，LCD 显示此透明电极形状。为防止液晶疲劳，提高液晶寿命，显示字符时应在透明电极和公共电极之间加 50~500Hz 的交变电场。通常用“异或”门来产生所需交变电场，如图 1-16 所示。

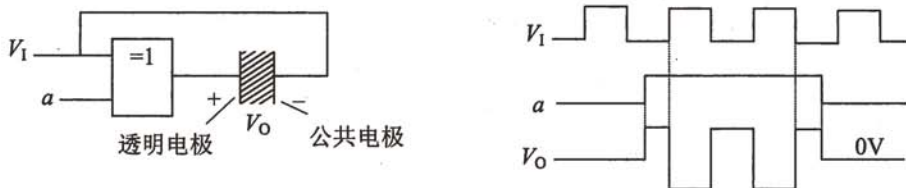


图 1-16 LCD 显示驱动电路及波形

## 5. 数据选择器和数据分配器

### (1) 数据选择器

数据选择器又称多路开关，它是以“与或”门或“与或非”门为主的电路。它可以在选择信号的作用下，从多个输入通道中选择某一个通道的数据作为输出。常见的数据选择器有二选一、四选一、八选一、十六选一等。

图 1-17 给出了一个二选一的数据选择器，有两个输入信号  $A$  和  $B$ ，一个输出值和一个选择信号  $S$ 。选择信号  $S$  决定了哪个输入量会成为输出值。该数据选择器的函数关系式可用下式表示：

$$C = (A \cdot \bar{S}) + (B \cdot S)$$

其对应的门电路实现可由图 1-17 实现。

数据选择器除有选择输入信号的功能外，还可利用它实现任意组合逻辑函数。例如四选一的

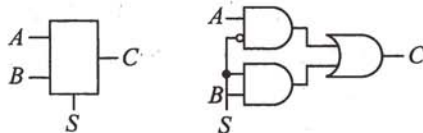


图 1-17 二选一数据选择器及其内部逻辑实现

变量的组合逻辑函数。图 1-18 是一个四输入多路选择器，其真值表如表 1-6 所示。

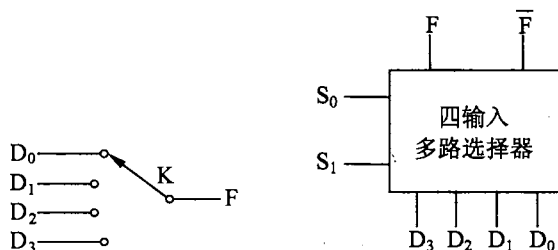


图 1-18 四输入多路数据选择器的实现

表 1-6 四输入多路选择器的真值表

选 择		数 据 输 入				输 出
$S_1$	$S_0$	$D_3$	$D_2$	$D_1$	$D_0$	$F$
0	0	x	x	x	0	0
0	0	x	x	x	1	1
0	1	x	x	0	x	0
0	1	x	x	1	x	1
1	0	x	0	x	x	0
1	0	x	1	x	x	1
1	1	0	x	x	x	0
1	1	1	x	x	x	1

多路选择器中也常设置使能端  $F$  或  $\bar{F}$ ，它的作用和译码器中的使用端相似，可用来扩展数据选择器的通路数，实现更多路的选择。

## (2) 数据分配器

数据分配器又称多路分配器，它有一个输入端和多个输出端，其逻辑功能是将一个输入端的信号送至多个输出端中的某一个，简称 DMUX，作用与 MUX 正好相反。

如图 1-19 (a) 所示为单刀四掷开关示意图，如图 1-19 (b) 所示为四位多路分配器的框图。该多路分配器的真值表如表 1-7 所示。

数据分配器的核心部分实际上是一个带有使能端的全译码器，可以把数据分配器理解为是输出受  $X$  控制的译码器。从表 1-7 中不难看出，当数据输入端  $X$  固定为 1 时，它就是一个 2-4 译码器。此时， $X$  就相当于译码器的使能端，而选择端  $S_1$ 、 $S_0$  就相当于译码器的输入端。



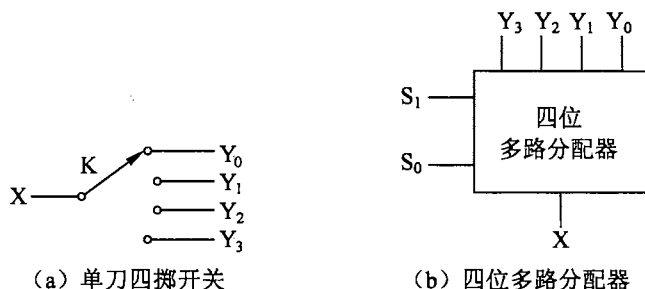


图 1-19 四位多路分配器

表 1-7 四位多路分配器的真值表

输 入	选 择		数 据 输 入			
X	$S_1$	$S_0$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	0	×	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

### (3) 双向多路开关

前述的多路选择器和多路分配器都是单向的，称为多路形状和反多路开关。在实际应用中，有时需要双向的多路开关，如果把多路选择器和多路分配器联用，就可以实现在一条线上分时地传送多路信号，即在相同地址输入的控制下，将多路输入信号的任一路从对应的一路输出。如图 1-20 所示是一个利用数据选择器和数据分配器实现的 8 位数据传输电路。

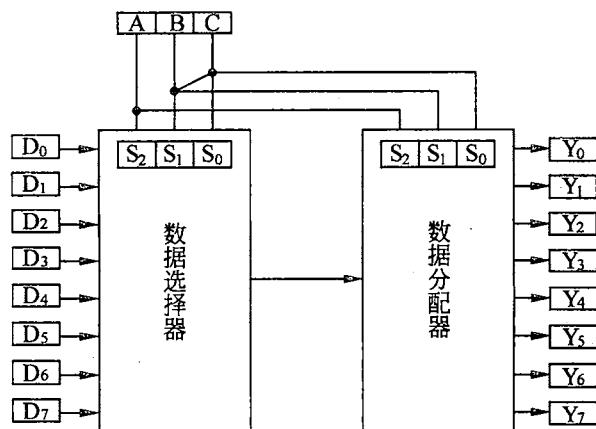


图 1-20 8 位数据传输电路

### 1.3.2 时序逻辑电路

所谓时序逻辑电路,是指电路任一时刻的输出不仅与该时刻的输入有关,而且还与该时刻电路的状态有关。因此,时序逻辑电路中必须包含记忆元件。触发器是构成时序逻辑电路的基础。常用的时序逻辑电路有寄存器和计数器等。

#### 1. 时钟信号

时钟信号是时序逻辑的基础,它用于决定逻辑单元中的状态何时更新。时钟信号是指有固定周期并与运行无关的信号量,时钟频率(Clock Frequency, CF)是时钟周期的倒数。如图 1-21 所示,时钟周期(Clockcycle Time)由两部分内容组成:高电平和低电平。时钟边沿触发信号(Edge-triggered Clocking)意味着所有的状态变化都发生在时钟边沿到来时刻。

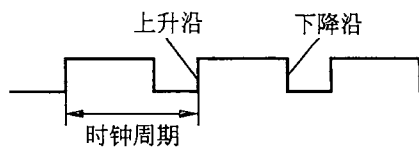


图 1-21 时钟信号

在边沿触发机制中,只有上升沿或下降沿才是有效信号,才能控制逻辑单元状态量的改变。至于到底是上升沿还是下降沿作为有效触发信号,则取决于逻辑设计的技术。

同步是时钟控制系统中的主要制约条件。同步就是指在有效信号沿发生时刻,希望写入单元的数据也有效。数据有效则是指数据量比较稳定(不发生改变),并且只有当输入发生变化时数值才会发生变化。由于组合电路无法实现反馈,所以只要输入量不发生变化,输出值最终会是一个稳定有效的量。

#### 2. 触发器

触发器种类很多。按时钟控制方式来分,有电位触发、边沿触发、主-从触发等方式。按功能分类,有 R-S 型、D 型、J-K 型等功能。同一功能触发器可以由不同触发方式来实现。对使用者来说,在选用触发器时,触发方式是必须考虑的因素。因为相同功能的触发器,若触发方式选用不当,系统达不到预期设计要求。这里将以触发方式为线索,介绍几种常用的触发器。

##### (1) 电位触发方式触发器

当触发器的同步控制信号 E 为约定“1”或“0”电平时,触发器接收输入数据,此时输入数据 D 的任何变化都会在输出 Q 端得到反映;当 E 为非约定电平时,触发器状态保持不变。鉴于它接收信息的条件是 E 出现约定的逻辑电平,故称它为电位触发方式触发器,简称电位触发器。

图 1-22 给出了被称为锁定触发器(又称为锁存器)的电位触发器的逻辑图。由 E 控制 D 和  $\bar{D}$  能否进入触发器:  $E=1$ , D 和  $\bar{D}$  进入基本触发器,此时  $Q=D$ ,  $\bar{Q}=\bar{D}$ ;  $E=0$ , D 和  $\bar{D}$  被封锁,触发器状态保持,以 E 的“1”电平撤除前 D 的最终电平决定 E=0 期间 Q 的状态。

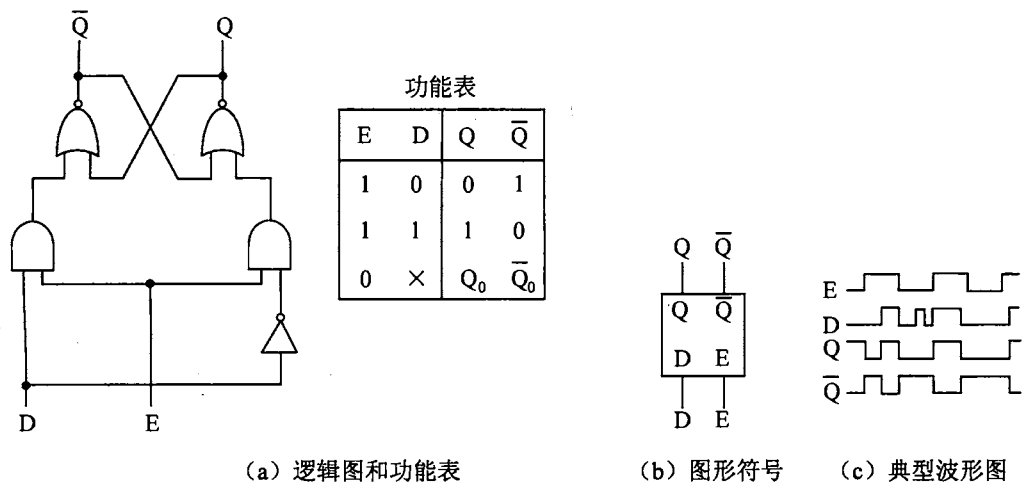


图 1-22 锁存器

电位触发器具有结构简单的优点。在计算机中常用它来组成暂存器。

## (2) 边沿触发方式触发器

具有如下所述特点的触发器称为边沿触发方式触发器简称边沿触发器。触发器是时钟脉冲 CP 的某一约定跳变（正跳变或负跳变）来到时的输入数据。在 CP=1 及 CP=0 期间以及 CP 非约定跳变到来时，触发器不接收数据。

常用的正边沿触发器是 D 触发器，图 1-23 给出了它的逻辑图及功能表。

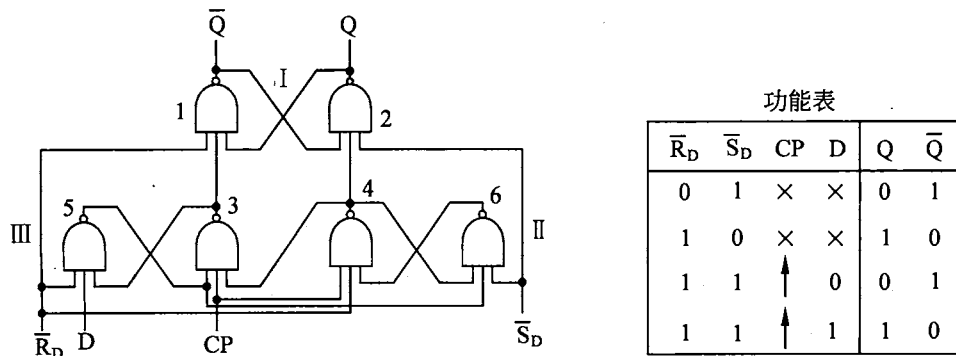


图 1-23 D 触发器逻辑图

下面比较边沿触发器和电位触发器。

电位触发器在 E=1 期间来到的数据会立刻被接收。但对于边沿触发器，在 CP=1 期间

来到的数据,必须“延迟”到该  $CP=1$  过后的下一个  $CP$  边沿来到时才被接收。因此边沿触发器又称延迟型触发器。

边沿触发器在  $CP$  正跳变(对正边沿触发器)以外期间出现在  $D$  端的数据和干扰不会被接收,因此有很强的抗数据端干扰的能力而被广泛应用,它除用来组成寄存器外,还可用来组成计数器和移位寄存器等。

至于电位触发器,只要  $E$  为约定电平,数据来到后就可立即被接收,它不需像边沿触发器那样保持到约定控制信号跳变到来才被接收。

### (3) 触发器的开关特性

描述触发器的参数很多,其中既有描述传输延迟的参数,也有描述各输入波形宽度要求的参数,还有描述各输入波形之间时间配合要求的参数。如果在使用时不能满足参数的要求,电路就不能正常地工作。

## 3. 寄存器与移位器

### (1) 寄存器

寄存器主要用来接收信息、寄存信息或传送信息,通常采用并行输入—并行输出的方式。由于一个触发器仅能寄存一位二进制代码,所以要寄存  $n$  位进制代码,就需要具备  $n$  个触发器。随着组成寄存器的触发器的触发方式不同,寄存器也有不同的触发方式,最常用的是正跳沿触发的  $D$  触发器,这种寄存器的各位在同一时刻( $CP$  脉冲的上升沿作用下)接收信息。也有一些寄存器的信息接收是通过电位信号(使能  $G$ )控制的,即高电平触发,这种寄存器又称为锁存器,其主要用途是把一些短暂的信号锁存(锁住并保存)起来,以达到时间上的扩展。

寄存器中除具有若干触发器以外,还应有门电路构成的控制电路,以保证信息的正确接收、发送和清除。

### (2) 移位寄存器

在时钟信号控制下,将所寄存的信息向左或向右移位的寄存器称为移位寄存器。按照信息移动方向的不同,移位寄存器可以分为单向(左移或右移)及双向移位寄存器。

按照信息的输入/输出方式不同,移位寄存器可以分为:串行输入—串行输出、串行输入—并行输出和并行输入—串行输出3种工作方式。从移位寄存器的外部特征来看,串行输入—串行输出的移位器仅需要一条数据输入线和一条数据输出线,而串行输入—并行输出的移位器需要一条数据输入线和多条数据输出线,并行输入—串行输出的移位器需要多条数据输入线和一条数据输出线。将串行输入信息变换成并行输出信息的过程,称为“串—并变换”,反之,将并行输入信息变换为串行输出信息的过程,称为“并—串变换”,这在计算机的接口电路中使用十分广泛。串入并出和并入串出移位寄存器的示意图如图1-24所示。

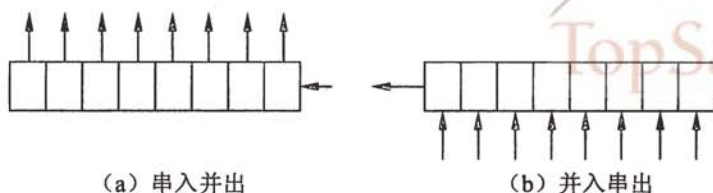


图 1-24 串入并出和并入串出移位寄存器

#### 4. 计数器

计数器是由各种触发器加逻辑门组成的，它的基本功能就是用来累计输入脉冲的个数。计数器不仅可以用来计数，也可用来定时和分频等。计数器的种类很多，按照组成计数器各触发器的状态转换所需的 CP 脉冲是否统一，可以分为同步计数器和异步计数器；按照计数值的增减情况，可以分为加法（递增）计数器、减法（递减）计数器；按照计数基数，可分为二进制计数器、十进制计数器。

##### (1) 异步计数器和同步计数器

异步计数器的特点是没有公共的时钟脉冲，除第一级外，每级触发器都是由前一级的输出信号触发的，所以高一级触发器的翻转有待低一级触发器翻转后才能进行。由于异步计数器的进位方式是串行的，故又称为串行计数器。

由于异步计数器是串行进位的，所以计数器总的延迟时间是各级触发器延迟时间之和，进位信号的传递时间限制了计数器的工作速度。另外，由于各触发器不是在同一时间翻转，因此各触发器输出之间存在着“偏移”，若对计数器输出进行译码，译码器输出就会出现“毛刺”，且计数器的倍数越多，偏移越大，“毛刺”越宽，可能会引起错误。

同步计数器的特点是各个触发器的时钟脉冲均来自同一个计数输入脉冲，各级触发器在计数脉冲作用下同时翻转（即并行进位），所以又称为并行计数器。同步计数器总的延迟时间与级数无关，因而速度可以得到提高。由于同步计数器需要把计数脉冲同时送到各个触发器的 CP 端，因此要求产生计数脉冲的电路具有较大的负载能力。

##### (2) 加法、减法、和可逆计数器

加法计数器的状态变化和数的依次累加相对应，减法计数器的状态变化和数的依次递减相对应，而可逆计数器既可实现加法计数又可实现减法计数，由控制信号选择相应状态的累加或递减。

##### (3) 计数器的“模”

计数器的计数值是用触发器的状态组合来表示的，在计数脉冲作用下使一组触发器的状态逐个转换成不同的状态组合来表示数的增加或减少，即可达到计数的目的。计数器在运行时，所经历的状态是周期性的，总是在有限个状态中循环，通常将一次循环所包含的

状态总数称为计数器的“模”。

二进制计数器的进位基数是 $2^r$ ,即2、4、8、16、...,也可称为模 $2^r$ 计数器,这类计数器相对比较简单;而非十进制计数器的各个状态虽然也是按照十进制计数规律编码的,但因为要去除某些不需要的状态,故这类计数器比较复杂。

### 1.3.3 总线电路及信号驱动

嵌入式计算机的总线系统提供微处理器、存储器及 I/O 设备之间的数据交换机制。要将存储器和其他外围设备加入到系统中,只需要将它们连接到总线系统上,并加入必要的解码逻辑电路即可。总线系统是由 CPU 控制的, CPU 把设备的地址放到地址总线上,再把总线控制信号放到控制总线上,设置数据传送方向和定时控制方法,从而实现 CPU 通过数据总线对设备的读写操作。

#### 1. 总线

总线实际上就是一组通信线路,在同一时刻,每条通信线路上能够传输一位用二进制表示的0或1的信号;在某一时间段内,每条通信线路可以传输一系列的二进制数字信号。如果一条总线上包含多条通信线路,可以同时传送多个二进制信号,则称为该总线为并行总线。如果一条总线上只包括用于接收和发送的1~2条通信线路,每次只传送一位二进制数据,则称该总线为串行总线。按总线所传送的信息类型可分为地址总线、数据总线和控制总线。

总线的性能由以下几个方面表示。

- 总线带宽:表示单位时间内,总线所能传输的最大数据量,一般用 MByte/s 表示。
- 总线宽度:通常把一条总线所包括的通信线路的数目的多少称为总线宽度。总线宽度通常有 8、16、32、64 位之分。在总线工作频率一定的条件下,在单位时间内总线传输数据量与总线宽度成正比。
- 总线的单元时钟频率:对于同步总线,采用统一的时钟脉冲作为总线定时基准。总线的时钟频率越高,总线上的数据操作越快。
- 总线的负载能力:指总线上可连接模块的最大数目。

由于数据总线是双向的,可以连接多个设备,如同时连接 ROM 和 RAM,这就存在总线冲突的可能性。如果两个设备正好同时把数据放到总线上,就可能发生总线冲突(Bus Collision)。总线冲突意味着两个设备的输出连到了一起,如果一个设备输出是高电平,而另外一个设备输出是低电平,那么在电源和地之间就会出现导通现象,使两个设备中的一个处于失效。因此,只有具有三态输出的设备才能够连接到数据总线上。当连接到总线上的设备不使用总线时,总线处于高阻状态,此时设备在物理上与总线“断开”,设备不能够向总线发送信息,避免干扰总线的正常操作,同时设备也不作为总线的负载,为总线可靠

传输信息创造有利条件。

## 2. 三态门

三态门 (ST 门) 主要用在应用于多个门输出共享数据总线。为避免多门输出同时占用数据总线, 这些门的使能信号 (EN) 中只允许有一个为有效电平 (如高电平), 由于三态门的输出是推拉式的低阻输出, 且不需接上拉 (负载) 电阻, 所以开关速度比集电极开路的门电路 (Open Collector Gate, OC 门) 快, 常用三态门作为输出缓冲器。

三态门是具有 3 种逻辑状态的门电路。这 3 种状态为: 逻辑 “0”, 逻辑 “1” 和浮动状态。所谓浮动状态, 就是三态门的输出呈现开路的高阻状态。三态门与普通门的不同之处在于, 除了正常的输入端和输出端之外, 还有一个控制端  $G$  (或  $\bar{G}$ )。只有当控制端有效时, 该三态门才满足正常的逻辑关系; 否则输出将呈现高阻状态, 相当于这个三态门与外界断开联系。根据输入/输出的关系和控制有效电平, 可以分成 4 种类型的三态门, 如图 1-25 所示。

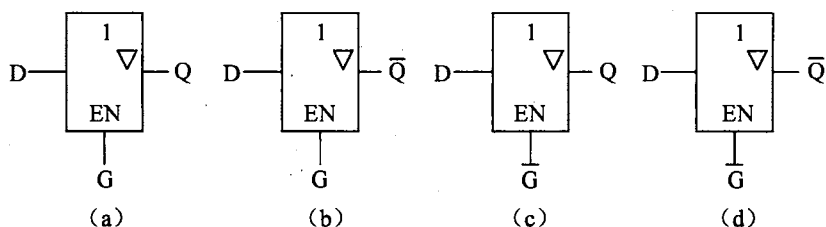


图 1-25 4 种类型的三态门

三态是微处理器系统信号的基本状态。当微处理器写数据时, 它会把需要写入的数据放在数据总线上, 并驱动总线进行写操作。此时, 与该总线连接在一起的其他无关设备都要进入三态状态, 以避免与写入设备发生冲突。同理, 当微处理器从数据总线上读取数据时, 它驱动总线信号进入三态状态, 同时, 向微处理器发送数据的设备把发送给微处理器的数据放入总线等待读取。三态状态可以使多个设备输出端共用一条总线, 但每一时刻只允许同一个设备对总线进行驱动。

图 1-26 给出了 3 个三态驱动器的工作情况。其中三态驱动器用图中的三角符号表示, 它的工作方式如下: 当它的选择信号有效时, 三态驱动器的输出和输入相同; 否则, 其输出处于浮动状态。图中, 当 Select A 信号有效, Select B 和 Select C 信号无效时, 输出将和三态驱动器 A 的输入相同, 而驱动器 B 和 C 都不会影响输出。因出, 输出与选择信号有效的三态驱动器保持一致。需要说明的地, 三态驱动器的选择信号的有效状态可以是高电平, 也可以是低电平。

对于图 1-26 所示电路, 当 3 个选择信号都无效时, 输出是不确定的, 称之为处于浮动状



态, 该信号状态是高是低, 或是处于高低之间的某种状态, 都不能确定, 它取决于该电路的瞬时状态, 从而使整个系统的状态无法预见。如果其输出连接到某个外部设备, 可能会使外部设备产生误动作。为了解决这个问题, 需要在输出端口增加一个电阻。如图 1-27 所示, 该电阻一端与电源  $V_{CC}$  相连, 一端与输出端口连接。当 3 个选择信号都处于无效状态时, 没有一个三态驱动器驱动输出信号, 这时在  $V_{CC}$  的作用下, 有电流流过电阻, 使输出端口的电压信号变高, 称该电阻为上拉 (pull-up) 电阻。同时, 如果该电阻另一端与地信号相连, 则当 3 个选择信号都处于无效状态时, 输出端口的电压信号会变低, 我们称这样的电阻为下拉 (pull-down) 电阻。

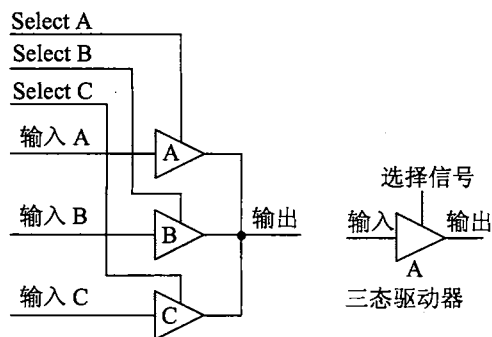


图 1-26 三态驱动器电路

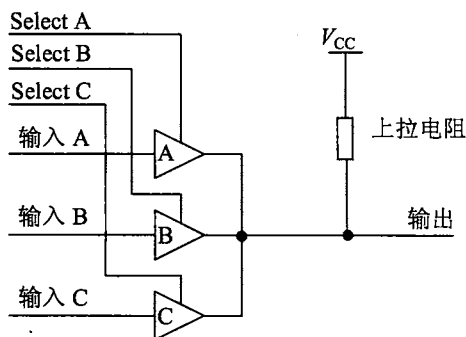


图 1-27 增加上拉电阻的三态驱动电路

三态设备可以分为两大类。一类是单向传输设备, 另一类是双向传输设备。单向传输设备只能在一个方向上进行数据输出。虽然这类设备的输出信号可以有 3 种状态 (高电平、低电平、高阻), 但其输出端不能同时做为输入端来使用。而双向传输设备输出端的 3 种状态同时可以作为输入端使用。微处理器和 RAM 的数据总线就是双向传输总线, 可以用来接收和发送数据; ROM 上的数据总线就是单向传输总线, 只有用于发送数据。

### 3. 总线的负载能力

所谓总线的负载能力即驱动能力, 是指当总线接上负载 (外围设备) 后不能影响总线输入/输出的逻辑电平, 例如 PC 总线中的输出信号, 在输出低电平要吸收电流 (由负载流入信号源), 以 IOL 表示, 这时的负载能力指当它吸收了规定电流时, 仍能保持逻辑低电平。输出高电平的负载能力以 IOH 表示, 这是一个由信号源流向负载的输出电流。当输出电流超过规定值时, 输出逻辑电平会降低, 甚至变到阈值以下。

当总线上所接负载超过总线的负载能力时, 必须在总线和负载之间加接缓冲器或驱动器, 最常用的是三态缓冲器, 其作用是驱动 (使信号电流加大, 可带动更多负载) 和隔离 (减少负载对总线信号的影响)。

#### 4. 单向和双向总线驱动器

74244 是一种单总线缓冲器/驱动器/接收器芯片,其内部结构如图 1-28 所示,输入/输出真值表如表 1-8 所示,其中 Z 表示高阻。当控制端  $\overline{G} = 0$  时,总线输出  $Y = A$ ; 当控制端  $\overline{G} = 1$  时,总线输出呈高阻状态。

表 1-8 74244 真值表

输 入		输 出
$\overline{G}$	A	Y
0	0	0
0	1	1
1	x	Z

74245 是一种双总线收发器,其内部结构如图 1-29 所示,输入/输出真值表如表 1-9 所示。当控制端  $\overline{G} = 0$  且  $DIR = 0$  时,数据从 B 端传送到 A 端; 当控制端  $\overline{G} = 0$  且  $DIR = 1$  时,数据从 A 端传送到 B 端; 当控制端  $\overline{G} = 1$  时,总线呈高阻状态。

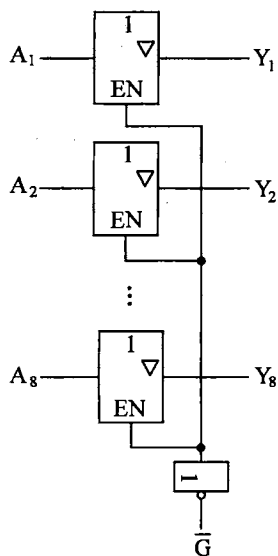


图 1-28 单向总线电路

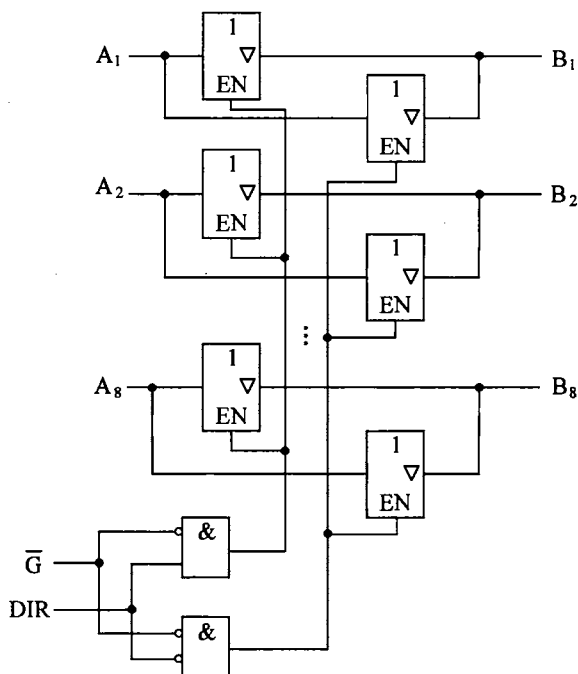


图 1-29 双向总线电路

表 1-9 74245 真值表

控制 输入		操 作
$\overline{G}$	DIR	
0	0	B 数据到 A 输出
0	1	A 数据到 B 输出
1	x	隔离

## 5. 总线复用

由于地址总线 and 数据总线要占用微处理器的大量引脚资源, 加上其他各种外部引脚信号, 使微处理器的尺寸和成本增加。采用总线复用技术 (multiplexed bus system) 可以实现数据总线和地址总线的共用, 这种方法可以有效地减少芯片的引脚, 从而降低芯片的成本。总线复用需要外加逻辑电路对总线信号进行解耦 (demultiplex)。如一个微处理器有 8 位数据总线和 8 位地址总线, 可以通过设置一条地址有效控制信号线来指示当前信号线上传送的是地址信号还是数据信号, 以实现数据总线与低地址总线的复用。当地址有效控制信号线发送地址有效信号时, 表示目前正在发送地址信号, 否则表示目前发送的是数据信号。

这样, 传送一个数据分成两个阶段。在第一阶段, 数据总线上传送地址总线的低 8 位, 同时在地址总线上传送地址总线的高 8 位, 此时, 连接到总线上的每一个设备都获得地址, 并判断是否为本设备的地址, 如果是, 则准备接收数据, 如不是, 则不予理睬。在第二阶段, 地址信号从总线上撤销, 同时地址有效控制信号线发送地址无效信号, 使总线被用于进行传输数据, 如图 1-30 所示。

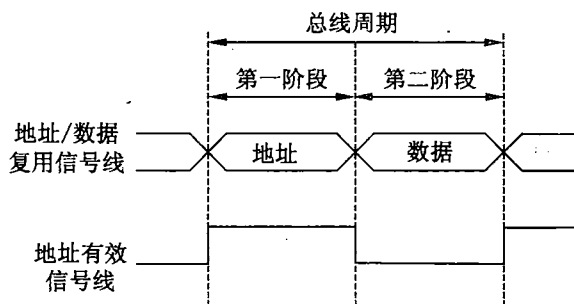


图 1-30 地址/数据总线分时复用信号

很显然, 总线复用带来两个问题:

- 需要增加外部电路对总线信号进行复用解耦。
- 由于占用总线通路, 复用总线的系统要比非复用总线系统的速度低。

## 6. 总线通信协议

总线数据通信方式按照传输定时的方法可分为同步式和异步式两类。

### (1) 同步通信

在同步方式中，总线上所有的事件发生都由时钟周期来定时。一个总线周期有多个时钟周期组成，连接到总线上的所有设备都通过时钟信号获取用于事件同步的时钟脉冲信号，所有的总线事件都应在一个时钟周期的开始时启动动作。

如图 1-31 所示，T1、T2、T3 表示总线周期，首先在第一个总线周期 T1 内，微处理器把从设备的地址放到地址总线上，同时将状态信息放到状态信号线上。一旦在地址总线和状态信号线上形成稳定的电平信号，微处理器发出地址有效信号，从设备收到地址有效信号后开始对地址总线上的地址进行译码。对于读操作，微处理器在第二个总线周期 T2 内通过读信号线发出读有效信号，从设备在下一个总线周期 T3 内，根据前面的地址译码确定的地址，将需要访问单元中的数据放到数据总线上；对于写操作，微处理器在总线周期 T2 内将数据放到数据总线上，等数据信号稳定后，微处理器通过写信号线发出写信号，在总线周期 T3 内把数据从数据总线上复制到目标单元中。

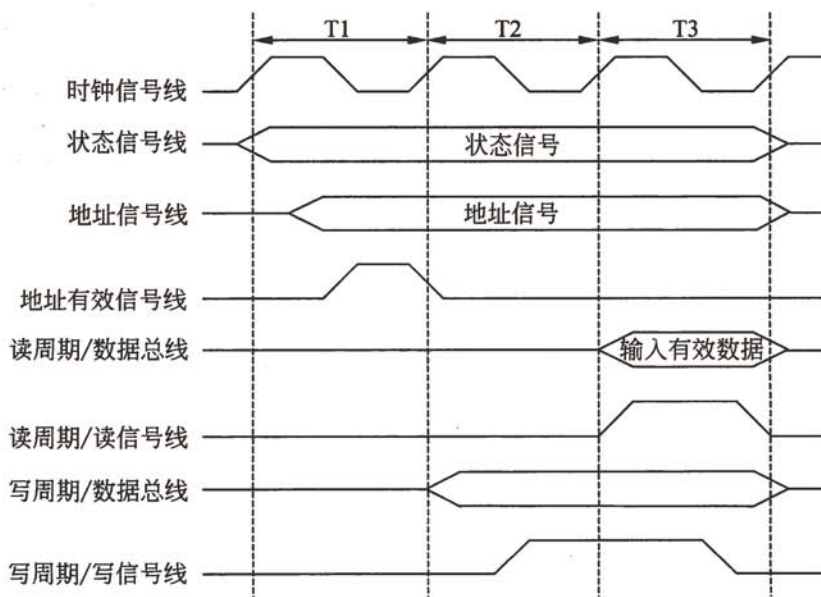


图 1-31 同步定时方式下的总线操作

### (2) 异步通信

另一种总线操作使用一个在 CPU 和设备之间的握手信号，去除了公共的时钟信号，从而使得操作成为异步的。两条握手信号分别称为“就绪”(ready)和“应答”(acknowledge)。在异步方式下，总线操作周期时间不是固定的，操作的每个步骤都有一个信号表示。异步

方式允许总线周期有较大的变化范围。根据握手信号的相互作用,异步通信方式可有非互锁、半互锁和全互锁3种可能的方式。在非互锁方式中,发送设备将数据放在总线上,延迟一定时间后发出就绪信号,通知对方数据已在总线上,接收设备根据这个就绪信号接收数据,并发出应答信号作回答,表示数据已接收到,发送设备收到应答信号后可以撤销数据,以便进行下一次传输。

上述握手信号发出后,经过固定的时间就自动撤销。如果总线上各设备之间的速度差异很大,这种方式比较简单,有利于提高传输速度,但有时不能保证就绪信号和应答信号正确到达对方。因为,当握手信号过短时,速度慢的设备容易将其错过,而当握手信号过长时则会延迟到下一个周期,使下一周期的握手信号产生错误。半互锁方式与非互锁方式类似,只是让就绪信号保持到发送设备接收到应答信号为止。

这样解决了就绪信号的宽度问题,但应答信号的宽度仍难以确定。在全互锁方式中,数据发送设备在发出数据后发出数据准备就绪信号,接收设备在接收到后发出应答信号,发送设备在收到应答后复位就绪信号,在就绪信号复位后接收设备才复位应答信号。这样,就绪信号和应答信号的宽度是依据传输情况而变化的,传输距离不同,信号的宽度也不同,从而解决了通信中的异步定时问题。

这种异步互锁式总线被广泛采用,因为它可适合各种工作速度的设备,总线周期是可变的,但它比较复杂,每次传输数据时需要传递4个握手信息,不利于提高传输速度。由于全互锁方式中的就绪信号和应答信号的上升边沿和下降边沿都是触发边沿,因此这种方式称为四边沿协议。异步总线写操作时序如图1-32所示。

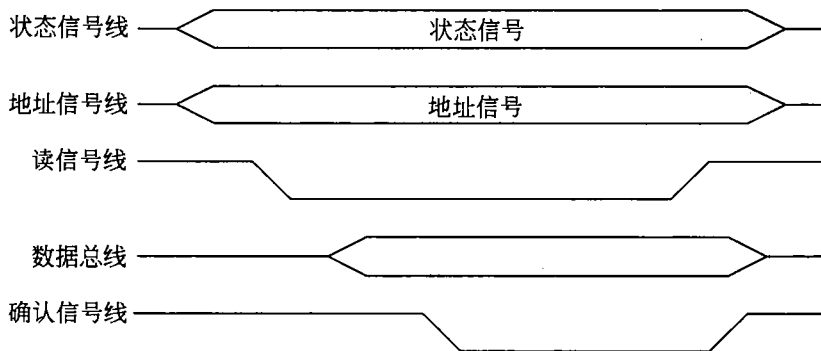


图 1-32 异步总线写操作时序

## 7. 总线仲裁

总线上的设备一般分为总线主设备和总线从设备。总线主设备是指具有控制总线能力的模块,通常指微处理器;与之相对应的总线从设备,是指能够对总线上的数据请求做出



被任一模块占用；其次才是有模块提出了总线请求。“允许”信号在链接的模块之间传输，直到提出总线“请求”的那个模块为止。这里用“允许”信号的边沿触发，它把共享总线的各模块要使用总线时，便发生信号禁止后面的部件使用总线。通过这种方式，就确定了请求总线各模块中优先级最高的模块。显然，在这种方式中，当优先级高的模块频繁请求时，优先级别低的模块可能很长时间都无法获得总线。一旦有模块占用总线后，“允许”信号就不再存在。

### (2) 并联优先级判别法

图 1-34 中有  $n$  个模块，都可作为总线主设备，每个模块都有总线“请求”线和总线“允许”线，模块之间是独立的，没有任何控制关系。这些信号接到总线优先控制器（仲裁器）。任一模块使用总线，都要通过“请求”线向仲裁器发出“请求”信号。仲裁器一般由一个优先级编码器和一个译码器组成。该电路接到某个模块或多个模块发来的请求信号后，首先优先级编码器进行编码，然后由译码器产生相应的输出信号，发往请求总线模块中优先级最高的模块，并把“允许”信号送给该模块。被选中的模块撤销总线“请求”信号，输出总线“忙”信号，通知其余模块，总线已经占用。在一个模块占用总线的传输结束以后，就把总线“忙”信号撤销，仲裁器也撤销“允许”信号。根据各请求输入的情况，仲裁器重新分配总线控制权。

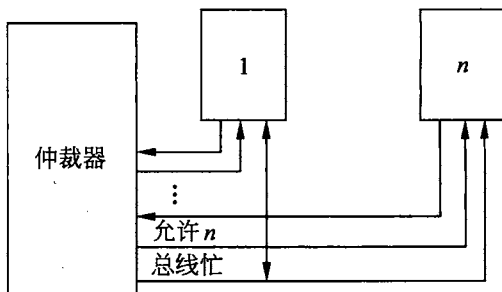


图 1-34 并联优先级判别法的示意图

### (3) 循环优先级判别法

循环优先级判别方法类似于并联优先级判别方法，只是其中的优先级是动态分配的，原来的优先级编码器由一个更为复杂的电路代替，该电路把占用总线的优先级在发出总线请求的那些模块之间循环移动，从而使每个总线模块使用总线的机会相同。

## 1.3.4 电平转换电路

数字集成电路具有体积小、重量轻、可靠性高、寿命长、功耗小、成本低和工作速度高等优点，因此它在现代电路设计中得到了广泛应用。目前数字集成电路种类繁多，功能各异。不同类型的集成电路在串接时，如果逻辑电平不兼容，并且考虑到负载能力的限制，中间需要串入接口电路，否则将引起逻辑混乱，或者损坏集成芯片。

### 1. 数字集成电路的分类

按照开关元件的不同，数字集成电路可以分为两大类：一类是双极型集成电路，采用晶体管作为开关元件，管内参与导电的有电子和空穴两种极性的载流子。另一类采用绝缘



栅场效应晶体管作开关元件,称为金属氧化物半导体(Metal-oxide Semiconductor, MOS)集成电路。这种管子内部只有一种载流子——电子或空穴参与导电,故又称单极型集成电路。MOS集成电路与双极型集成电路比较,具有很多优点,如制造工艺简单、集成度高、功耗低等,特别适宜于制造大规模集成电路。它的主要缺点是工作速度比较低。

### (1) 双极型集成电路

晶体管-晶体管逻辑电路(Transistor-Transistor Logic, TTL)是目前双极型数字集成电路中用得最多的一种。它具有比较快的开关速度、比较强的抗干扰能力以及足够大的输出幅度,并且带负载能力也比较强,所以得到了最为广泛的应用。

在双极型数字集成电路中,除了TTL电路以外,还有二极管-三极管逻辑(Diode-Transistor Logic, DTL)、高阈值逻辑(High Threshold Logic, HTL)、发射极耦合逻辑(Emitter Coupled Logic, ECL)和集成注入逻辑(Integrated Injection Logic, IIL)等几种逻辑电路。

DTL是早期采用的一种电路结构形式,它的输入端是二极管结构而输出端是三极管结构。因为它的工作速度比较低,所以不久便被TTL电路取代了。

HTL电路的特点是阈值电压比较高。当电源电压为15V时,阈值电压达7~8V。因此,它的噪声容限比较大,有较强的抗干扰能力。它的主要缺点是工作速度比较低,所以多用在对工作速度要求不高而对抗干扰能力要求较高的一些工业控制设备中。目前它几乎完全为CMOS电路所取代。

ECL电路中的三极管工作在非饱和状态,是一种非饱和电路,有极高的工作速度,此外它还具有输出阻抗低,带负载能力强,电路内部开关噪声低,使用方便灵活等优点。它的主要缺点是:噪声容限低,电路功耗大,输出电平的稳定性较差。目前ECL电路主要用于高速、超高速数字系统中。

IIL电路结构简单,集成度高,功耗低,缺点是输出电压幅度小,抗干扰能力较差,且工作速度低。目前IIL电路主要用于制作大规模集成电路的内部逻辑电路,很少用来制作中、小规模集成电路产品。

### (2) MOS集成电路

按照所用MOS管类型的不同,可分为3种:由PMOS管构成的PMOS集成电路;由NMOS管构成的NMOS集成电路;由PMOS管和NMOS管构成的互补(Complementary)MOS集成电路,简称CMOS。PMOS和NMOS组件中各只含有一种MOS管,习惯上称它们为MOS集成电路,以与CMOS集成电路相区别。

PMOS集成电路问世较早,但由于其速度低,现已很少使用;NMOS集成电路速度稍高,且直流电源电压较低,在工艺上可以制造出开启电压较低的器件,故NMOS集成电路仍在使用中。CMOS电路由于其静态功耗极低,工作速度较高,抗干扰能力强,故被广泛采用。

## 2. 常用数字集成电路逻辑电平接口技术

虽然 TTL 电路具有很多优点,但它毕竟不能满足生产中不断提出来的各种特殊要求,例如高速、高抗干扰、低功耗等,因而在工程中还经常用到 ECL、CMOS 等数字集成电路。在微机测控系统中,习惯于用 TTL 电路作为基本电路元件,根据需要可能采用 CMOS、ECL 等芯片,因此存在 TTL 电路与这些数字电路的接口问题。

ECL 的特点是速度快,但抗干扰性能差,功耗也高;TTL 的应用广泛,成本低廉,有许多种类可供选择;CMOS 功耗最低,抗干扰性能优良,不仅适用于中、小规模集成电路,而且在大规模集成组件中应用也很普遍。

下面讨论 TTL 与 ECL 和 CMOS 之间的电平转换接口。

### 1) TTL 与 ECL 电平转换接口

ECL 电路电压一般为 $-5.2\text{V}$  (CE10K 系列),逻辑高电平输出为  $V_{OH} = -0.9\text{V}$ ,低电平为 $-1.75\text{V}$ ;对 CE100K 系列则电源电压为 $-4.5\text{V}$ ,输出高电平为  $V_{OH} = -0.955\text{V}$ ,低电平  $V_{OL} = -1.705\text{V}$ 。

#### (1) TTL→ECL 转换

利用集成芯片 CE1024 即可完成 TTL 到 ECL 的电平转换。它有一个公共的选通脉冲输入端 B,若 B 为低电平,ECL 的所有 Y 为低电平, $\bar{Y}$  为高电平。

#### (2) ECL→TTL 转换

CE10125 为四 ECL—TTL 电平转换器,它的输入与 ECL 电平兼容,具有差分输入和抑制 $\pm 1\text{V}$  共态干扰输入能力,输出是 TTL 电平。如果有某路不用时,须将其一个输入端接到 VBB 端上,以保证电路的工作稳定性。

在小型系统中,ECL 和 TTL 可能均使用  $+5\text{V}$  电源,此时需用分立元件来实现接口。

### 2) TTL 与 CMOS 电平转换接口

CMOS 反相器当其使用电源电压为  $5\text{V}$  时,输出低电平电压最大值为  $0.05\text{V}$ ,高电平最小值为  $4.95\text{V}$ ,输出低电平电流最小为  $0.5\text{mA}$ ,高电平电流最小为  $-0.5\text{mA}$ ;对于带缓冲门的 CMOS 电路,当供电电源电压为  $5\text{V}$  时, $V_{IL} \leq 1.5\text{V}$ , $V_{IH} \geq 3.5\text{V}$ 。而对于不带缓冲门的 CMOS 电路, $V_{IL} \leq 1\text{V}$ , $V_{IH} \geq 4\text{V}$ 。

#### (1) TTL→CMOS 转换

由于 TTL 电路输出高电平的规范值为  $2.4\text{V}$ ,在电源电压为  $5\text{V}$  时,CMOS 电路输入高电平  $V_{IH} \geq 3.5\text{V}$ 。这样就造成了 TTL 与 CMOS 电路接口上的困难。解决的办法是在 TTL 电路输出端与电源之间接一上拉电阻 R。上拉电阻 R 的取值由 TTL 的高电平输出漏电流  $I_{OH}$  来决定,不同系列的 TTL 应选用不同的 R 值。一般有:

- 74 系列,  $4.7\text{k}\Omega \geq R \geq 390\Omega$
- 74H 系列,  $4.7\text{k}\Omega \geq R \geq 270\Omega$

- 74L 系列,  $27\text{k}\Omega \geq R \geq 1.5\text{k}\Omega$
- 74S 系列,  $4.7\text{k}\Omega \geq R \geq 270\Omega$
- 74LS 系列,  $12\text{k}\Omega \geq R \geq 820\Omega$

如果 CMOS 电路的电源电压高于 TTL 电路的电源电压。同时 CMOS 电路应使用具有电平移位功能的电路, 如 CC4504, CC40109 及 BH017 等, 至于 CMOS 电路的电源电压可在 5~15V 范围内任意选定。

## (2) CMOS→TTL 转换

关于 CMOS 到 TTL 的接口, 由于 TTL 电路输入短路电流较大, 就要求 CMOS 电路在 VOL 为 0.5V 时能给出足够的驱动电流, 因此需使用 CC4049、CC4050 等作为接口器件。

## 3) CMOS 与晶体管和运放的接口

### (1) CMOS 与晶体管的接口

利用 CMOS 驱动晶体管, 可以达到驱动较大负载的功能。如图 1-35 所示, 由  $R_1$ 、 $R_2$  提供晶体管  $T_1$  的导通电平, 利用  $T_2$  实现电流放大, 从而驱动负载  $R_L$  工作,  $R_L$  可以是继电器、显示灯等器件。

图中  $R_1$  的取值可由下式决定:

$$R_1 = \frac{V_{OH} - (V_{BE1} + V_{BE2})}{I_B + (V_{BE1} + V_{BE2})/R_2}$$

$V_{OH}$  为 CMOS 输出高电平,  $V_{BE1}$ 、 $V_{BE2}$  为晶体管  $T_1$ 、 $T_2$  BE 极之间的正向压降, 其值通常可取 0.7V;  $R_2$  是为改善电路开关性能而引入的。其值一般取 4~10k $\Omega$ 。

### (2) CMOS 与运放的接口

图 1-36 为 CMOS 与运放的接口电路。其中图 (a) 为运放与 CMOS 电路电源独立时的接口; 图 (b) 为 CMOS 与运放使用同一电源时的接口电路。

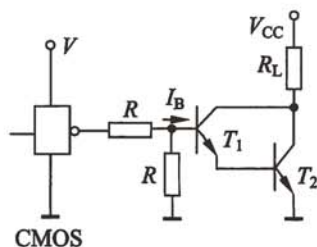


图 1-35 CMOS 与晶体管接口

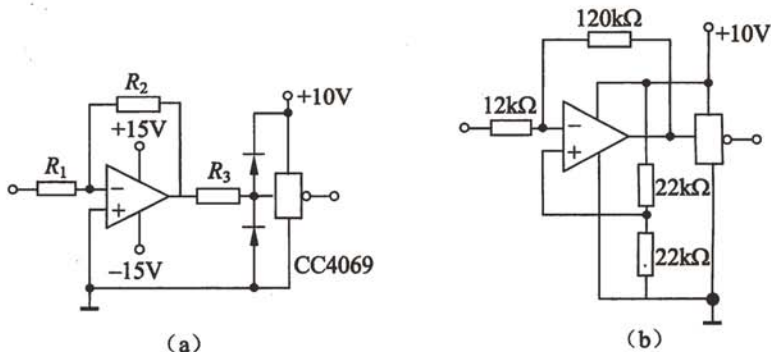


图 1-36 CMOS 与运放接口

### 1.3.5 可编程逻辑器件基础

#### 1. 可编程逻辑器件 (Programmable Logic Device, PLD) 概述

随着微电子技术的发展,设计与制造集成电路的任务已不完全由半导体厂商来独立承担。系统设计师们更愿意自己设计专用集成电路 (Application Specific Integrated Circuit, ASIC) 芯片,而且希望 ASIC 的设计周期尽可能短,最好是在实验室里就能设计出合适的 ASIC 芯片,并且立即投入实际应用之中,因而出现了现场可编程逻辑器件 (Field Programmable Logic Device, FPLD),其中应用最广泛的当属现场可编程门阵列 (Field Programmable Gate Array, FPGA) 和复杂可编程逻辑器件 (Complex Programmable Logic Device, CPLD)。

早期的可编程逻辑器件只有可编程只读存储器 (PROM)、紫外线可擦除只读存储器 (EPROM) 和电可擦除只读存储器 (EEPROM) 3 种。由于结构的限制,它们只能完成简单的数字逻辑功能。

其后,出现了一类结构上稍复杂的可编程芯片,即可编程逻辑器件 (PLD),它能够完成各种数字逻辑功能。典型的 PLD 由一个“与”门和一个“或”门阵列组成,而任意一个组合逻辑都可以用“与-或”表达式来描述,所以,PLD 能以乘积和的形式完成大量的组合逻辑功能。

这一阶段的产品主要有可编程阵列逻辑 (Programmable Array Logic, PAL) 和通用阵列逻辑 (Generic Array Logic, GAL)。PAL 由一个可编程的“与”平面和一个固定的“或”平面构成,“或”门的输出可以通过触发器有选择地被置为寄存状态。PAL 器件是现场可编程的,它的实现工艺有反熔丝技术、EPROM 技术和 EEPROM 技术。还有一类结构更为灵活的逻辑器件是可编程逻辑阵列,它也由一个“与”平面和一个“或”平面构成,但是这两个平面的连接关系是可编程的。PLA 器件既有现场可编程的,也有掩膜可编程的。在 PAL 的基础上,又发展了一种通用阵列逻辑,如 GAL16V8 和 GAL22V10 等。它采用了 EEPROM 工艺,实现了电可擦除、电可改写,其输出结构是可编程的逻辑宏单元,因而它的设计具有很强的灵活性,至今仍有许多人使用。这些早期的 PLD 器件的一个共同特点是可以实现速度特性较好的逻辑功能,但其过于简单的结构也使它们只能实现规模较小的电路。

为了弥补这一缺陷,20 世纪 80 年代中期,Altera 和 Xilinx 公司分别推出了类似于 PAL 结构的扩展型复杂可编程逻辑器件和与标准门阵列类似的现场可编程门阵列,它们都具有体系结构和逻辑单元灵活、集成度高以及适用范围宽等特点。这两种器件兼容了 PLD 和通用门阵列的优点,可实现较大规模的电路,编程也很灵活。与门阵列等其他专用集成电路相比,它们又具有设计开发周期短、设计制造成本低、开发工具先进、标准产品无需测试、质量稳定以及可实时在线检验等优点,因此被广泛应用于产品的原型设计和产品生产(一

一般在 10 000 件以下) 之中。几乎所有应用门阵列、PLD 和中小规模通用数字集成电路的场合均可应用 FPGA 和 CPLD 器件。

## 2. PLD 的电路表示法

PLD 表示法在芯片内部配置和逻辑图之间建立了一一对应关系, 并将逻辑图和真值表结合起来, 形成一种紧凑而又易于识读的表达式。

### (1) 连接方式

PLD 电路由与门阵列和或门阵列两种基本的门阵列组成。图 1-37 是一个基本的 PLD 结构图。门阵列交叉点上连接有 3 种方式。

- 硬线连接: 硬线连接是固定连接, 不能用编程加以改变。
- 编程接通: 是通过编程实现接通的连接。
- 可编程断开: 通过编程已使该处连接呈断开状态。

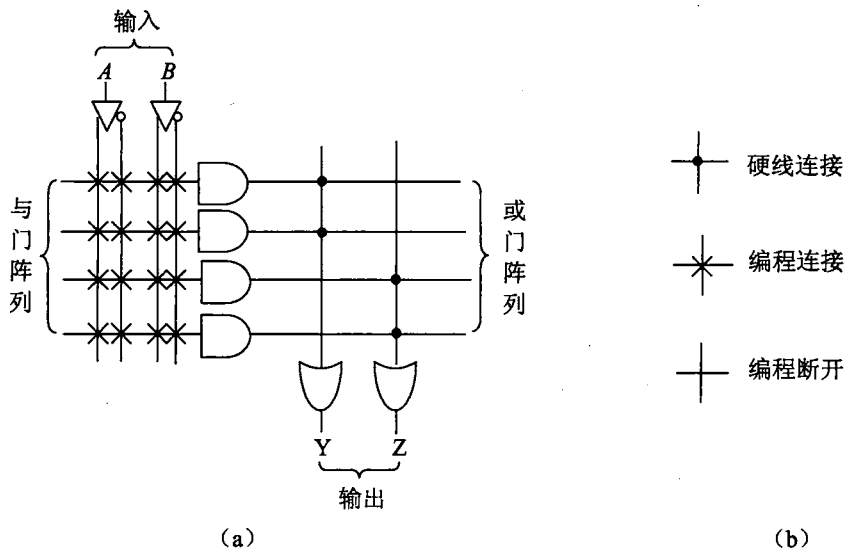


图 1-37 PLD 表示法

### (2) 基本门电路的 PLD 表示法

几种基本门在 PLD 表示法中的表达式如图 1-38 所示。一个四输入与门在 PLD 表示法中的表示如图 1-38 (a) 所示,  $L_1 = ABCD$ , 通常把  $A$ 、 $B$ 、 $C$ 、 $D$  称为输入项,  $L_1$  称为乘积项 (简称积项)。一个四输入或门如图 (b) 所示, 其中  $L_2 = A+B+C+D$ 。缓冲器有互补输出, 如图 (c) 所示。

### (3) PROM 的 PLD 表示法

可编程的只读存储器实质上可以认为是一个可编程逻辑器件, 它包含一个固定连接的

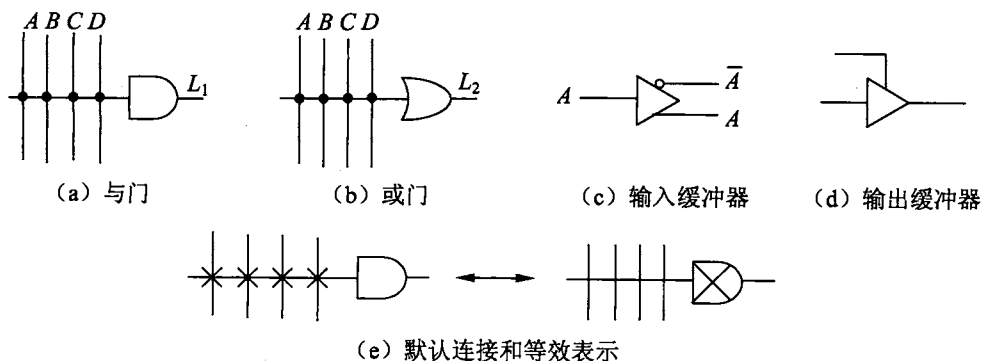


图 1-38 基本门的 PLD 表示法

与门阵列（即全译码的地址译码器）和一个可编程的或门阵列。图 1-39 是四位输入地址码四位字长 PROM 的 PLD 表示法表示。图中可编程或阵列的可编程单元都以编程断开连接形式表示，图 (b) 为其等效表示。

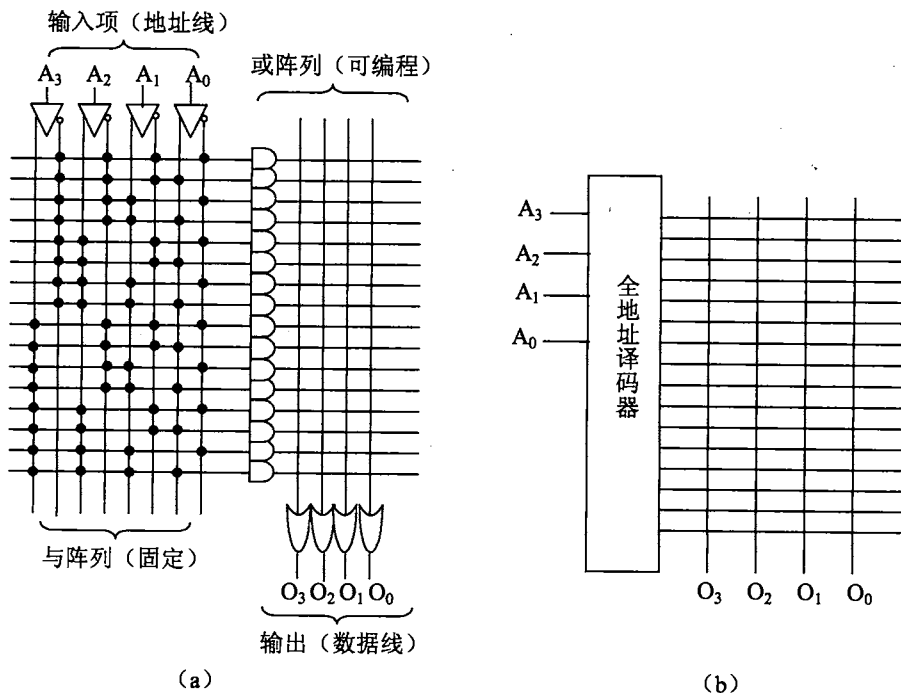


图 1-39 PROM 的 PLD 表示法

### 3. 可编程阵列逻辑器件 (PAL) 和可编程逻辑阵列 (PLA)

可编程阵列逻辑器件 PAL 采用可编程与门阵列和固定连接的或门阵列的基本结构形式。用 PAL 门阵列实现逻辑函数时, 每个函数是若干个乘积项之和, 但乘积项数目固定不变 (乘积项数目取决于所采用的 PAL 芯片)。PAL 编程前的结构图如图 1-40 (a) 所示, 图中与门阵列的可编程单元用 “+” 表示, 省略了可编程连接符 “x”; 图 1-40 (b) 给出了编程后的简化结构图, 图中用 “+” 表示可编程单元断开连接, 用 “ $\otimes$ ” 表示编程连接, 以示

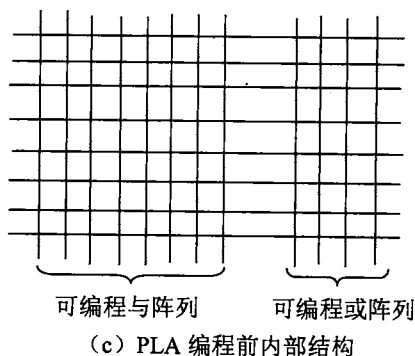
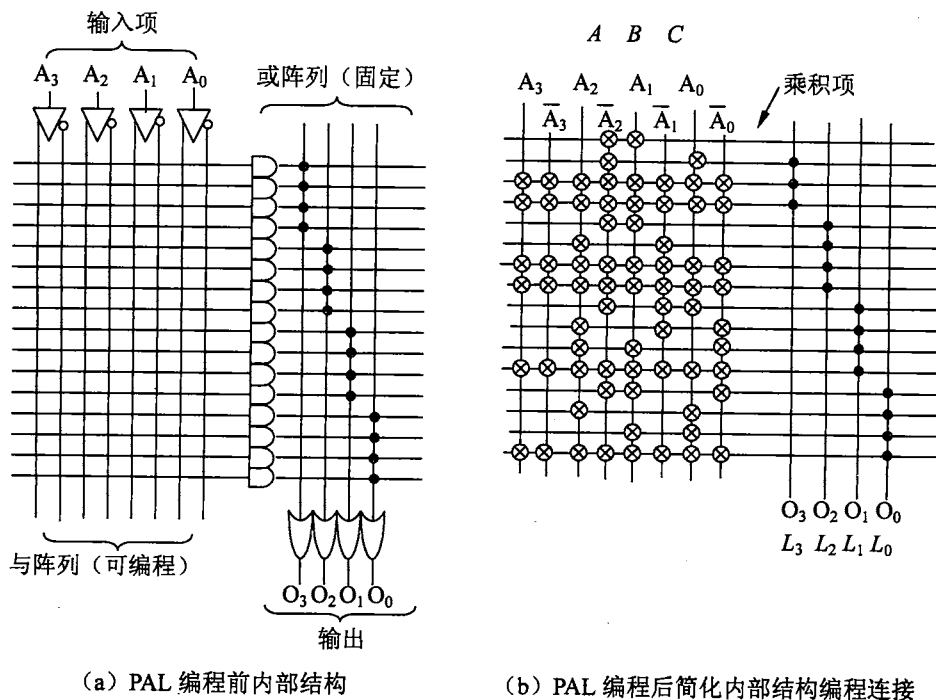


图 1-40 PAL 和 PLA 的基本结构



与或门阵列的固定连接相“.”区别。

由图 1-40 (a) 可知, 每个或门有固定的 4 个输入 (与门的输出, 即乘积项), 每个与门都有 8 个输入端 (与 4 个输入变量相对应), 所以, 该 PAL 每个输出 (函数) 有四乘积项, 每个乘积项最多可含有 4 个输入变量。

编程前与门的 8 个输入和 4 个输入变量及其反变量接通, 这是与门阵列的默认状态。编程后, 有些连接被熔断, 从而获得需要的乘积项。默认状态时, 与门输出为 0。

图 1-40 (b) 中, 4 个输出函数分别为:

$$L_0 = \overline{A}B\overline{C} + AC + BC$$

$$L_1 = \overline{A}\overline{B}C + A\overline{B}\overline{C} + AB\overline{C}$$

$$L_2 = \overline{A}B + A\overline{B}$$

$$L_3 = \overline{A}B + \overline{A}C$$

可编程逻辑阵列 PLA 的与门阵列和或门阵列都是可编程的, 使用更灵活。图 1-40 (c) 给出 PLA 的简化图。

实际应用中的 PAL 芯片乘积项可有 8 个, 变量数可达 16 个, 如型号为 PAL16L8 可编程阵列逻辑器件。

#### 4. 可编程通用阵列逻辑器件 (GAL)

可编程通用阵列逻辑器件 GAL 是在 PAL 基础上发展起来的新一代逻辑器件, 它继承了 PAL 的与-或阵列结构, 又利用灵活的输出逻辑宏单元 OLMC 来增强输出功能。

##### 1) GAL 的基本结构

图 1-41 为可编程通用阵列逻辑器件 GAL16L8 内部逻辑结构及相应管脚分布。它由 5 部分组成:

- 8 个输入缓冲器 (引脚 2~9 作为输入)。
- 8 个输出缓冲器 (引脚 12~19 作为输出缓冲器的输出)。
- 8 个反馈/输入缓冲器 (将输出反馈给与门阵列, 或将输出端用作为输入端)。
- 可编程与门阵列 (由 8×8 个与门构成, 形成 64 个乘积项, 每个与门有 32 个输入, 其中 16 个来自输入缓冲器, 另 16 个来自反馈/输入缓冲器)。
- 8 个输出逻辑宏单元 (OLMC12~19, 或门阵列包含其中)。

除以上 5 个组成部分外, 该器件还有一个系统时钟 CK 的输入端 (引脚 1)、一个输出三态控制端 OE (引脚 11)、一个电源  $V_{CC}$  端 (引脚 20) 和一个接地端 (引脚 10)。

##### 2) 输出逻辑宏单元 (OLMC)

OLMC 的组成如下所示:

##### (1) 乘积项多路选择器 PTMUX

每个 OLMC 有 8 个乘积项作为输入, 其中第 1 个乘积项作为 PTMUX 的输入, 其他 7

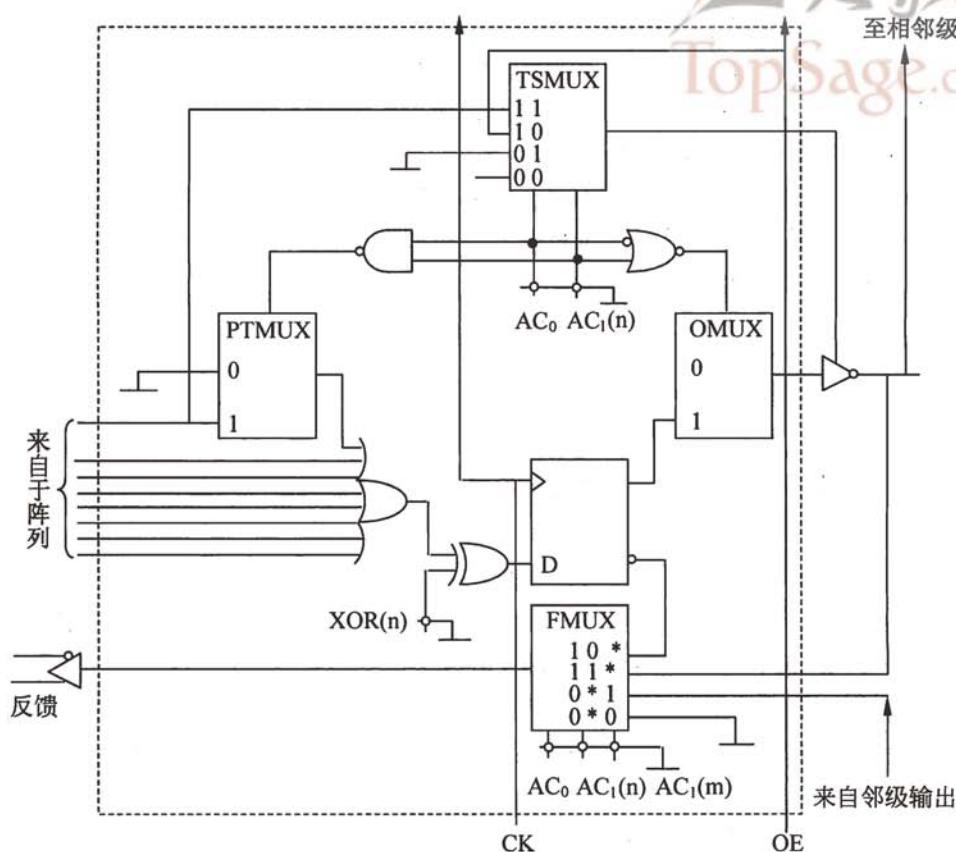


图 1-41 输出逻辑宏单元 OLMC

个乘积项为或门（或阵列的一部分）的输入。PTMUX 的另一输入为 0。当  $AC_0AC_1(n)$  为 11 时，PTMUX 输出为“0”；为其他值时，PTMUX 输出第 1 个乘积项。

### (2) D 触发器

在时钟 CK 作用下，接收异或门输出的信号，当异或门的 XOR(n) 输入为 1 时，接收或门输出的反码；当 XOR(n) 为 0 时，接收或门输出的原码，即由 XOR(n) 值决定输出是高电位有效或低电位有效。

### (3) 输出选择多路器 OMUX

它的两个输入分别来自 D 触发器的输出端和异或门的输出端。当  $AC_0AC_1(n)$  为 10 时，选择 D 触发器输出（寄存器型输出）；为其他值时，选择异或门输出（组合逻辑输出）。

### (4) 输出允许控制选择多路器 TSMUX

当  $AC_0AC_1(n)$  为 00 时，输出为  $V_{CC}$  (1)，打开输出三态门；当  $AC_0AC_1(n)$  为 01

时, 输出为地(0), 输出三态门置于高阻态输出。 $AC_0AC_1(n)$  为 11 或 10 时, 分别将第一个输入乘积项或 OE 端送来的信号去控制输出三态门。

#### (5) 反馈源多路选择器 FMUX

该多路器在  $AC_0$  和  $AC_1(n)$  的控制下, 选择“0”、本级的输出、D 触发器  $\bar{Q}$  或来自邻级的输出之一反馈到输入端作为与阵列的输入信号。

GAL 电路使用灵活, 且具有多次擦除功能, 特别适合于进行新产品试制的试验。

### 5. 门阵列 (Gate Array, GA)

门阵列设计利用预先制造好的“母片”来进行图设计。母片上通常以一定的间距成行成列的排列着基本单元的电路。芯片内部区域上每一个小方块代表一个基本单元。基本单元一般由 6~10 个晶体管组成, 基本单元内所含的电路元件(晶体管、电阻等)是事先制备好的, 所有基本单元内部元件的排列都是相同的, 因而所有基本单元的大小和形状都是一样的。对基本单元内部元件进行不同连线, 就要构成各种类型的门电路或触发器。这样, 门阵列设计系统应该有一个单元库, 它存放各种门及触发器的内部连线以及输入、输出端的引线位置等信息。由于门阵列基本单元所包含的晶体管数目比较少, 因此一个基本单元所能组成的单元的功能是比较简单的。如图 1-42 所示, 各个小方块之间的区域是通道区, 基本单元之间的连线在通道区内行走。基本单元区的四周是芯片的输入输出(I/O)单元, 用作输入输出电平配合、输入保护、输出功率驱动等, 芯片的最外围是压焊点区。

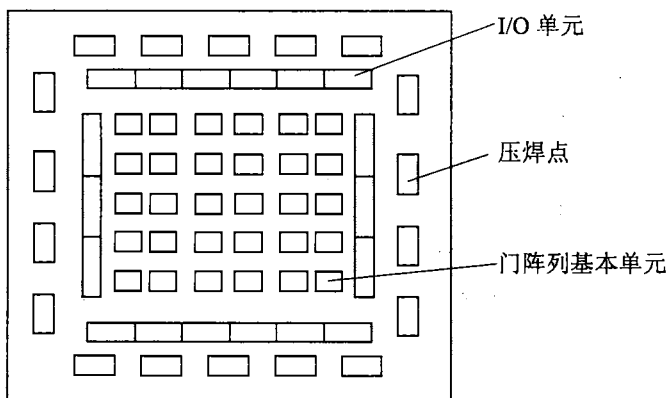


图 1-42 门阵列芯片布图

门阵列设计的优点是设计自动化程度较高, 设计周期短, 设计成本低。因为母片已完成了整个集成电路制造工艺的大部分流程。当用户提交了逻辑图之后, 只要进行基本单元内部布线和基本单元之间的互连就可以了。因此把这种器件称为半用户器件或半定制器件。

门阵列的缺点是布图密度低, 这是因为基本单元所能构成的逻辑元件功能比较简单,

并且品种有限,为了使所有单元间的边线能布通,势必造成芯片面积利用率的下降,一般情况下,利用率不超过 70%~80%。

## 6. 可编程程序门阵列 (Programmable Gate Array, PGA)

可编程程序门阵列主要由 4 个部分组成:

- 可编程序逻辑宏单元 (CLB)。它以阵列形式分布在芯片的中心部位。每个 CLB 由若干个触发器及一些可编程序组合逻辑部件组成。CLB 可通过编程来实现用户所需的逻辑。
- 可编程序输入输出宏单元 (IOB)。它排列在 CLB 四周,是芯片内部 CLB 与芯片外部引脚间的可编程接口,每个 IOB 可进行边沿触发器、锁存器、上拉电阻选择、三态选择等输入输出方式控制。IOB 也是通过编程来实现所需的输入输出方式控制的。
- 互连资源。它包括可编程的互连开关矩阵、内部长线、总线等。
- 重构逻辑的程序存储器。它以阵列形式分布在整个芯片上。PGA 器件工作时,首先要将用户所需实现的逻辑以某种程序形式从片外读至 PGA 重构逻辑的程序存储器内,该程序存储器的存储单元输出直接去控制指定的 CLC 和 IOB 等单元,从而使器件有确定的功能。常把这一过程称为配置。

美国 XILINX 公司的 XC 系列 PGA 按其集成度有多种型号。其中, XC-3090 型产品,每个芯片的逻辑能力(即可用门电路数)为 9 000, CLB 的块数为 320, IOB 的块数为 144,程序存储器的程序位为 64160。

下面对 CLB、IOB 及重构逻辑程序存储器分别进行介绍。

### (1) 可编程逻辑宏单元 (CLB)

不同公司的 PGA 产品系列,它们的 CLB 中所含触发器数目和可编程序组合逻辑的功能是不同的。图 1-43 为 XILINX 公司 XC-3090 型 PGA 的 CLB 逻辑图,该 CLB 内包含两个正沿 D 触发器、一个组合逻辑功能块以及内部控制电路。它的输入、输出端分别为:5 个逻辑变量输入 a, b, c, d, e, 数据输入  $d_i$ , 时钟输入 k, 时钟使能输入 ec, 复位输入  $r_d$ , x 和 y 是 CLB 的输出。

组合逻辑功能块有两个输出: F 和 G。功能块可能的输入为外部输入 a, b, c, d, e 和 2 个触发器的输出:  $Q_x$  和  $Q_y$ 。功能块可有 3 种构成方式:

- 可构成两个独立输出的逻辑功能子块,每个子块最多可有 4 个变量输入,即可从 a~e,  $Q_x$ ,  $Q_y$  中进行如图 1-44 (a) 所示的选择。
- 可构成一个 5 变量输入、单输出的功能块,些时  $F = G$ , 如图 1-44 (b) 所示。
- 可实现 7 变量输入、单输出组合逻辑,其中变量 e 用来对上、下两功能子块的输出进行选择,如图 1-44 (c) 所示。

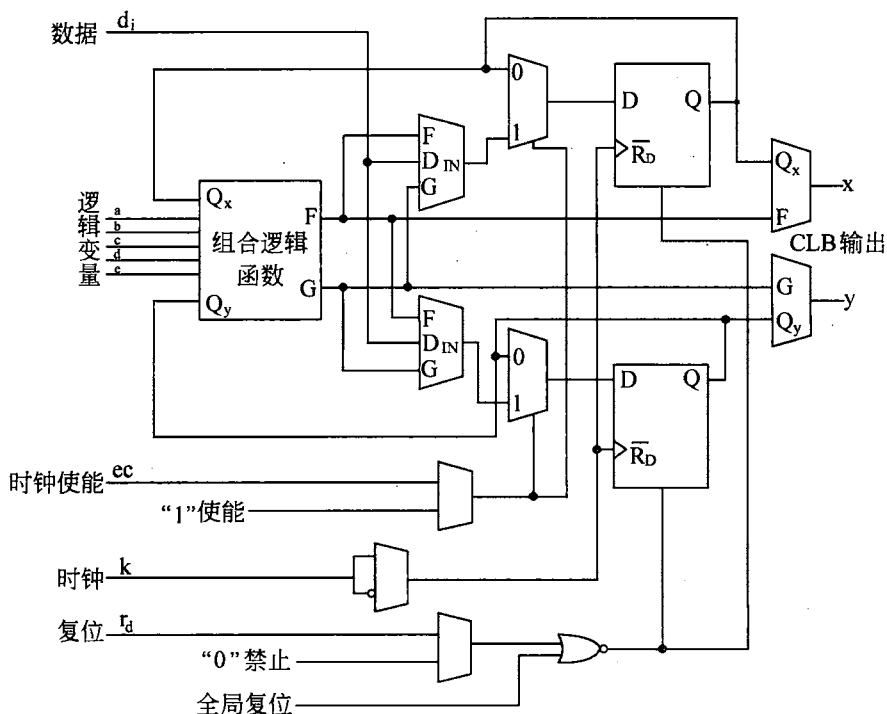


图 1-43 可编程逻辑宏单元 CLB 逻辑图

CLB 内两个触发器的数据输入可以从外部数据输入  $d_i$ 、组合逻辑功能块的输出  $F$  或  $G$ 、触发器的输出  $Q$  中选择。可以选择外部时钟  $k$  和  $\bar{k}$  作为触发器的时钟。触发器的直接置“0”信号  $\bar{R}_D$  可以来自外部的复位输入  $r_d$ ，也可在器件进行配置时，由内部产生的全局复位信号进行复位。

CLB 的输出  $x$  可以从  $F$  或从触发器输出  $Q_x$  中选择； $y$  可从  $G$  或  $Q_y$  中选择。

下面通过一个有并行输入数据功能的 3 位二进制计数器为例，来说明如何用 CLB 实现给定逻辑。

一个用正沿 D 触发器的 3 位十进制计数器的 D 端逻辑表达式为：

$$D_0 = \bar{Q}_0$$

$$D_1 = Q_0 \oplus Q_1$$

$$D_2 = (Q_0 Q_1) \oplus Q_2$$

其中注脚“0”表示最低位。由该表达式，并考虑并行输入数据，可得计数器的逻辑图及功能表。其中，异或门及其输入与门是实现上述计数功能的表达式的；触发器 D 端的数据选择器是对异或门输出或者是对外部并输入数据  $A, B, C$  作出选择的。L 和 CE 分别

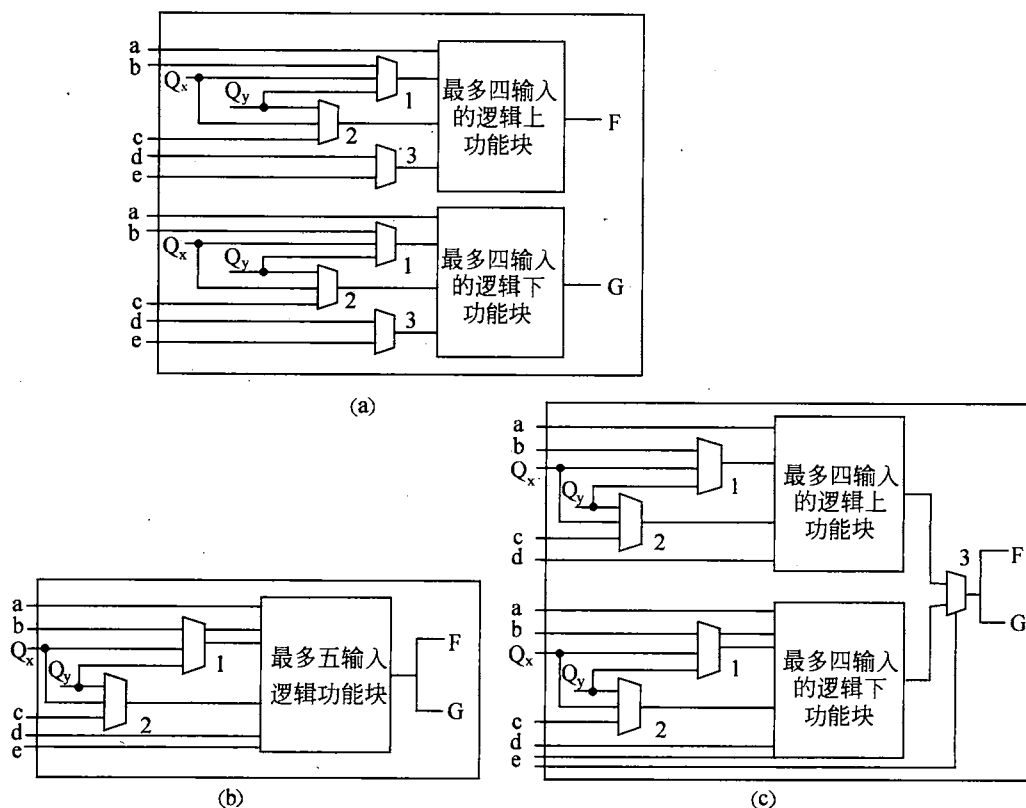


图 1-44 CLB 内组合逻辑功能模块的 3 种构成方式

是并行输入命令和计数命令。Carry 是计数器的进位输出，当  $CE = 1$ ，且  $Q_0 \sim Q_2$  均为“1”时，Carry=1。

各 CLB 的构成格式如表 1-10 所示。

表 1-10 CLB 的构成

CLB 块名	输 入									输 出	
	a	b	c	d	e	$d_i$	ec	k	$r_d$	x	y
0	CE	L	$Q_2$	$Q_1$	A	—	—	CK	—	Carry	$Q_0$
1	$Q_0$	CE	—	L	B	—	—	CK	—	—	$Q_1$
2	$Q_1$	$Q_0$	C	CE	L	—	—	CK	—	—	$Q_2$

下面先讨论按第一种构成方式来组合逻辑块的 CLB。CLB<sub>0</sub> 中的上逻辑功能块实现四输入与门功能。此时让 a 和 CE 相连；d 和 CLB<sub>1</sub> 的 y 输出（即  $Q_1$ ）相连，且使图 1-44 (a)

中的选择器 3 选择 d; 使 c 和  $CLB_2$  的 y 输出 (即  $Q_2$ ) 相连, 且使图 1-44 (a) 中选择器 2 选择 c; 使图 1-44 (a) 中选择器 1 选择  $Q_y$  (即  $Q_0$ ), 这样  $CLB_0$  的 F 输出即为  $Carry = CE \cdot Q_0 \cdot Q_1 \cdot Q_2$ 。使  $CLB_0$  中的下逻辑功能块实现和  $FF_0$  相连异或门及数据选择器。此时让 A 和 e 相连, 且图 1-44 (a) 中选择器 3 选择 e; 使 b 和 L 相连, 且选择器 1 选择 b; 选择器 2 选择  $Q_y$  (即  $Q_0$ )。这样, 就完成了  $CLB_0$  的构成。 $CLB_0$  的构成格式示于表 1-10。

### (2) 可编程输入输出宏单元

可编程输入输出宏单元有四种输入方式: 直接输入; 边沿触发器/锁存器输入; 直接输入方式中又可有 CMOS/TTL 电平输入和上拉电阻输入。

有 5 种输出方式: 直接输出/寄存器输出; 反相/不反相输出; 三态/导通态/截止态输出; 三态反相输出; 边沿速率全速输出/限速输出。

### (3) 重构逻辑的程序存储器

前面已经讲过, PGA 有一个重构逻辑存储器。 $CLB$  及  $IOB$  的编程数据是由存放在重构逻辑存储器中的程序来确定的。程序由 XILINX 公司的开发系统产生并直接装至重构逻辑存储器内的。开发系统能自动完成将用户逻辑划分为  $CLB$  和  $IOB$  的集合; 自动确定每个宏单元块所要承担的逻辑功能; 自动在逻辑块间进行互设计; 最终自动生成构成逻辑程序。开发软件包可在 PC-386 计算机上进行。也就是说,  $CLB$  及  $IOB$  是通过编程序来实现用户所需逻辑的, 而不是用如熔丝等元件实现逻辑编程序, 也不是像门阵列、宏单元阵列那样, 通过集成电路工厂的制作工艺来完成互连的。这样, PGA 可以允许用户多次修改逻辑, 且程序修改和装入方便, 比起门阵列和宏单元阵列来, PGA 更适合在产品试验或生产指不大时使用, 因为这样做, 更为经济和快捷。

## 1.4 嵌入式系统中信息表示与运算基础

### 1.4.1 进位计数制与转换

人们常用的是十进制数制, 但计算机中为便于存储及物理实现, 采用了二进制数制。二进制数的基数为 2, 只有 0、1 两个数码。为了书写和阅读的方便, 一般使用十六进制数来表示二进制数, 十六进制数的基数为 16, 数码有: 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F。本教材中, 通常用数字后面跟一个英文字母来表示该数的数制。十进制数用 D (Decimal) 或省略, 二进制数用 B (Binary), 十六进制数用 H (Hexadecimal) 来表示。二进制数、十进制数、十六进制数之间的数码转换如表 1-11 所示。



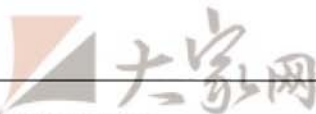


表 1-11 二进制数、十进制数、十六进制数之间的数码转换表

二进制数	十进制数	十六进制数	二进制数	十进制数	十六进制数
0000	0	0	1001	9	9
0001	1	1	1010	10	A
0010	2	2	1011	11	B
0011	3	3	1100	12	C
0100	4	4	1101	13	D
0101	5	5	1110	14	E
0110	6	6	1111	15	F
0111	7	7	10000	16	10
1000	8	8			

## 1.4.2 计算机中数的表示

计算机中的数是用二进制数来表示的，在某些情况下，要处理的数全是正数，此时就没有符号问题，如表示地址的数，为无符号数。8 位无符号数的表示的数的范围是 0~255，16 位无符号数的表示的数的范围是 0~65 535。

带符号数与无符号数的处理是有差别的，数的符号也是用二进制表示的，一般用最高有效位来表示数的符号，正数用 0 表示，负数用 1 来表示。为了运算方便常用原码、补码和反码表示机器中的数。8 位二进制原码表示的数的范围是 -127~+127，16 位二进制原码表示的数的范围是 -32 767~+32 767。原码表示的数比较直观，与真实值转换方便。

正数的反码表示与原码相同，负数的反码表示为该数的原码除符号位外按位取反。补码表示法中，正数的补码仍与原码相同，即数的最高有效位为 0 表示为正数，其余几位则表示该数的数值。一个负数的补码，最高有效位为 1，其余几位按原码各位求反，最末位加 1。例如，一个十进制数 +98 (十六进制 62H) 用八位字长表示的二进制原码为：01100010B，其反码和补码也均为 01100010B。而 -98 的二进制原码为 11100010B，反码是 10011101B，补码为 10011110B。

### 1. 数的定点和浮点表示

在一般书写中，小数点是用记号“.”来表示的，但在计算机中表示任何信息只能用 0 或 1 两种数码，如果计算机中的小数点用数码表示，则与二进制数位又不易区分。所以在计算机中小数点就不能够用记号表示，那么在计算机中小数点又如何确定呢？

为了确定小数点的位置，在计算机中，数的表示有两种方法：定点表示法和浮点表示法。所谓定点与浮点是指一个数的小数点位置是固定的还是浮动的。

#### (1) 定点表示法

所谓定点表示法，是指在计算机中所有数的小数点的位置人为约定固定不变。这样，

小数点的位置就不必用记号“.”表示出来了。一般地说,小数点可约定为固定在任何数位之后,但常用下列两种形式。

- 定点纯小数: 约定小数点位置固定在符号之后,如图 1-45 (a) 所示,如果假设定长为  $n$ ,则定点小数表示范围为  $1-2^{n-1} \sim -(1-2^{n-1})$ 。
- 定点纯整数: 约定小数点位置固定在最低数值位之后,如图 1-45 (b) 所示,如果假设字长为  $n$ ,则定点整数的表示范围为: 从  $2^{n-1}-1 \sim -(2^{n-1}-1)$ 。

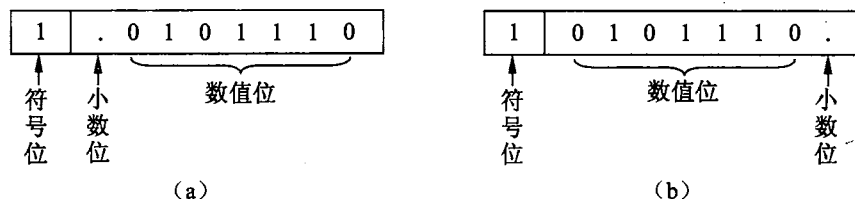


图 1-45 数的定点表示

显然,定点数表示法使计算机只能处理纯整数或纯小数,限制了计算机处理数据的范围。为了使计算机能够处理任意数,我们事先要将参加运算的数乘上一个“比例因子”,转化成纯小数或纯整数后进行运算。运算结果比例因子还原成实际数值。比例因子要取得合适,使参加运算的数、运算的中间结果以及最后结果都在该定点数所能表示的数值范围之内。

## (2) 浮点表示法

在浮点表示法中,小数点的位置是浮动的。为了使小数点可以自由浮动,浮点数由两部分组成,即尾数部分与阶数部分。浮点数在计算机中的表示方法如图 1-46 所示。

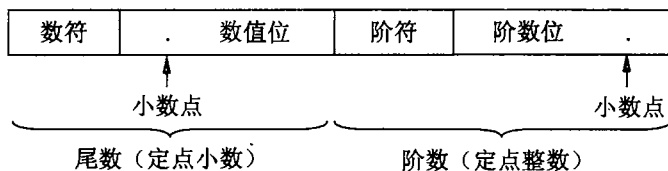


图 1-46 浮点数在机器中的表示方法

其中,尾数部分表示浮点数的全部有效数字,它是一个有符号位的纯小数;阶数部分指明了浮点数实际小数点的位置与尾数(定点纯小数)约定的小数点位置之间的位移量  $P$ 。该位移量  $P$ (阶数)是一个有符号位的纯小数。

当阶数为  $+P$  时,则表示小数点向右移动  $P$  位;当阶数为  $-P$  时,则表示小数点向左移动  $P$  位。因此,浮点数的小数点随着  $P$  的符号和大小而自由浮动。

从上述可知,一个浮点数是由两个定点数组合而成的,而一个定点数也可以看成是浮

点数的一个特例，即当浮点数的阶数部分为 0（表示浮点数实际小数点的位置与定点小数约定位置一致），这样，浮点数就只剩下尾数部分。同理，定点数表示法是浮点数表示法的基础，而浮点数表示法是定点数表示法的应用。它们之间的相互关系，从理论上看出下述关系。

我们知道，任意一个二进制数总可以表示为纯小数（或纯整数）和一个 2 的整数次幂的乘积。例如，任意一个二进制数  $N$  可写成：

$$N = 2^P \times S$$

式中  $S$  称为数  $N$  的尾数， $P$  称为数  $N$  的阶码，2 称为阶码的底。这里  $P$  和  $S$  都是用二进制表示的数。尾数  $S$  表示数  $N$  的全部有效数字，显然  $S$  采用的数位越多，则数  $N$  表示的数值精确度越高。阶码  $P$  指明了小数点的位置。显然  $P$  采用的数位越多，则数  $N$  表示的数值范围就越大。

如假定  $P=0$ ，此时， $N=S \times 2^0=S$ 。若尾数  $S$  为纯小数，这时数  $N$  为定点小数。

如假定  $P=0$ ，此时若尾数  $S$  为纯整数，则数  $N$  为定点整数。

如假定  $P$ =任意整数，此时，数  $N$  需要尾数  $S$  和阶数  $P$  两部分共同表示，即数  $N$  为浮点数。

显然，浮点数表示的数值范围比定点数表示的数值范围大得多。设浮点数的阶数位数为  $m+1$  位，尾数的位数为  $n+1$  位，则浮点数的取值范围为：

$$2^{-n} \cdot 2^{1(2^m-1)} \leq |N| \leq (1-2^{-n}) \cdot 2^{+(2^m-1)}$$

虽然浮点数具有表示数值范围大的突出优点，但是，浮点数的运算较为复杂。当计算机进行一次浮点数运算时，需要分别进行两次定点数运算。

例如，设两个浮点数为：

$$N_1 = 2^{P_1} \times S_1$$

$$N_2 = 2^{P_2} \times S_2$$

如  $P_1 \neq P_2$ ，则这两个数不能直接相加、减，必须首先将小数点对齐（即对阶）后，才能进行尾数间的加、减运算。对阶时，小阶向大阶看齐，即把小阶的小数点左移，在计算机中尾数数码右移，右移 1 位，阶码加 1，直至两数的阶码相同为止，然后两数才能相加减。

浮点数的乘除法，阶码和尾数须分别进行运算。

为了使计算机运算过程中不丢失有效数字，提高运算的精度，一般都采用二进制浮点规格化数。所谓浮点规格化，是指尾数  $S$  的绝对值小于 1 而大于或等于 1/2，即小数点后面的一位必须是“1”。例如， $N=2+100 \times 0.1011101$  就是一个浮点规格化数。

由于浮点数运算复杂，运算器中除了尾数运算部件外，还有阶码运算部件，控制部件也相应地复杂了，故浮点机的设备增多，成本较高。

在计算机中, 究竟采用浮点制还是定点制, 必须根据使用要求设计。目前, 一般小型机、微型机多采用定点制, 而大型机、巨型机及高档微型机中多采用浮点制。

### 1.4.3 非数值数据编码

嵌入式系统不仅能够对数值数据进行处理, 还能够对文本和其他的非数值数据信息进行处理。非数值数据是指不能进行算术运算的数据。除字符之外, 非数值数据还包括汉字、声音等。

#### 1. 字符和字符串的表示方法

字符包括: 大小写英文字母、数字、运算符、标点符号等。这些字符在计算机里必须用二进制数来表示, 目前微机中最常用的是美国信息交换标准代码 (American Standard Code for Information Interchange, ASCII 码)。这种代码用一个字节来表示一个字符, 共有 128 个字符。

##### (1) ASCII 字符编码

常见的 ASCII 码用 7 位二进制表示一个字符, 包括 10 个十进制数字 (0~9)、52 个英文大写和小写字母 (A~Z, a~z) 34 个专用符号和 32 个控制符号, 共计 128 个字符。在 128 个字符中有 96 个是可打印字符。

在计算机中, 通常用一个字节来存放一个字符。对于 ASCII 码来说, 一个字节右边的 7 位表示不同的字符代码, 而最左边一位可以作奇偶校验位, 用来检查错误, 也可以用于西文字符和汉字的区分标识。

ASCII 字符编码表如表 1-12 所示。由表中可见, 数字和英文字母都是按顺序排列的, 只要知道其中一个的十进制代码, 不要查表就可以推导出其他数字或字母的十进制代码。另外, 如果将 ASCII 码中 0~9 这 10 个数字的十进制代码去掉最高三位“011”, 正好与它们的十进制值相同, 这不但使十进制数字进入计算机后易于压缩成 4 位代码, 而且也便于进一步的信息处理。

表 1-12 ASCII 字符编码表

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	,	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u

续表

	000	001	010	011	100	101	110	111
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M	]	m	}
1110	RO	RS	.	>	N	↑	n	~
1111	SI	US	/	?	O	_	o	DEL

除标准 ASCII 字符编码外,有些公司通过不同的选择来使用高位 ASCII 的字符(这些的代码值从 80H~FFH),这种字符编码被称为扩展 ASCII 码。扩展 ASCII 码用八位二进制表示一个字符,一共可表示 256 个不同的字符。

## (2) 字符串的存放

字符串是指一串连续的字符。通常,它们在存储器中占用一片连续的空间,每个字节存放一个字符代码。字符串的所有字符在物理上是邻接的,这种字符串的存储方法称为向量法。例如,字符串 IFX>0 THEN READ (C),在字长为 32 位的存储器中的存放格式如图 1-47 (a) 所示。图中每一个主存单元可存放 4 个字符,整型字符串需 5 个主存单元。在每个字节中实际存放的是相应字符的 ASCII 码,如图 1-47 (b) 所示。

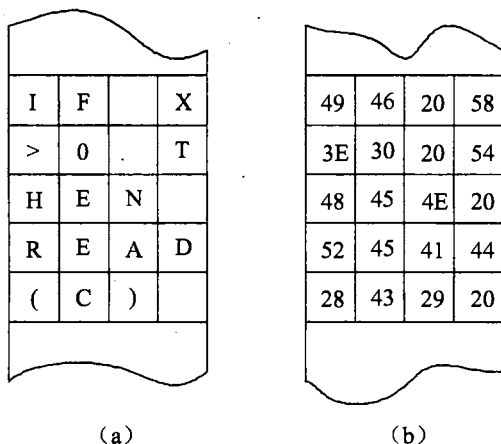


图 1-47 字符串的向量存放方案

字符串的向量存放法是最简单、最节省存储空间的方法。但是，当字符串需要进行删除和插入操作时，在删除或插入字符后面的子字符串需要全部重新分配存储空间，将花费较多的时间。为了克服向量存放法的缺点，另一种字符串的存储方法——串表法应运而生了。在这种存储方法中，字符串的每个字符代码后有一个链接字，用以指出下一个字符的存储单元地址。串表法不要求串中的各个字符在物理上相邻，原则上讲，串中各字符可以安排在存储器的任意位置上。在对字符串进行删除和插入操作时，只需修改相应字符代码后面的链接字即可，所以非常方便。但是，由于链接字占据了存储单元的大部分空间，使主存的有效利用率下降。例如，一个主存单元有 32 位，仅存放一个字符代码，而链接字占用了 24 位，这时，存放字符串信息的主存有效利用率只占 25%，这是串表法的最大缺点。

## 2. 汉字的表示方法

汉字处理技术是我国计算机推广应用工作中必须要解决的问题。汉字的字数繁多，字型复杂，读音多变，常用汉字有 7000 个左右。

根据对我国汉字使用频度的研究，可把汉字分为高频字（约 100 个）、常用字（约 3000 个）、次常用字（约 4000 个）、罕见字（约 8000 个）和死字（约 4500 个）5 种。

即正在使用的汉字字种达 15000 余个。根据我国 1981 公布的《通信用汉字字符集（基本集）及其交换码标准》GB2312-80 方案，把高频字、常用字和次常用字归纳为汉字基本字符集，共 6763 个字。按出现的频度分为：一级汉字 3755 个，二级汉字 3008 个，还有西文字母、数字、图形符号等 682 个，再加上专用汉字和符号，共 7445 个，这么多的汉字，其输入成为人们必须花大力气研究的课题，研究输入设备从 4 个方面展开。

### • 整字大键盘输入

整字大键盘是仿照旧式中文打印机，把汉字按部阵或习惯排列起来，用光笔选字的输入设备。它比较直观，但由于尺寸所限，所列出的汉字量只有一级汉字。同时在输入中英文混写的文件时，输入速度很慢。

### • 手写输入

手写输入是正在开发的一种输入方法，当汉字笔画很少时，速度较快；当笔画较多时，速度也很慢，并且要求书写要工整。

### • 语音输入

这是目前正在开发中的一种输入方法，它适合于联想及按惯用的修辞方式所表达的意思完整的句子。诗词、人名、地名以及单个字的输入比较困难，而且由于每个人的发音的特殊性，说话的音调、速度、情绪等都会影响输入的准确性。

### • 小键盘输入

小键盘输入，即是用标准英文键盘输入，这是目前人们已经花费了大量精力研究的一种输入方法。目前已提出的小键盘输入方法约有 2000 种左右，但总体上可分为如下几种：



- ◆ 汉字拼音输入码，如全拼码、双拼码等。
- ◆ 汉字字形编码，如五笔字型码、首尾码、101 码等。
- ◆ 汉字音形编码。
- ◆ 汉字数字编码，如区位码、电报码等。

小键盘输入的基本原理是：内码与字符一一对应，而外码（输入码）与内码具有多对一的关系。即对于内码可以有不同的输入方法，或者说，每一种汉字输入程序的基本功能，都是将输入码转换成内码。汉字的内码是计算机内部处理和存储汉字时使用的代码，内码有许多种，但以我国制定推行的 GB2312—80 国家标准信息交换用汉字编码（简称国标码）为最好。国标码采用两个字节 7 位汉字编码方式表示一个汉字，每个字节只使用后 7 位。这种设计虽使汉字与英文字符完全兼容，但当英文字符与汉字混合存储时，容易发生冲突。所以只能把国标码的两个字节的最高位（最左端一位）置 1，或者把其中一个字节的最高位置 1 后，作为汉字的内码使用。如此，汉字的内码既兼容英文 ASCII 码，又不会与 ASCII 码产生二义性，同时汉字与国标码具有很简单的一一对应关系。

当某一种输入码输入一个汉字到计算机之后，汉字管理模块立刻将它转换成 2 字节长的 GB2312—80 国标码，同时将国标码的每个字节的最高位置为 1，作为汉字的标识符，即将国标码转换为机器内部的代码——汉字内部码。

如：“啊”的国标码是：0011 0000 0001 0010 (3012H)

生成的汉字内码为： 1011 0000 1001 0010 (B0A1H)

除此之外，使用较多的内码，还有 HZ 码（国标码的变形）和大五码（Big5）等。

无论用那一种输入码输入汉字，在计算机内部存储时，都采用机内码，这也就是用一种汉字输入法输入的文档，也可以用另一种输入法对其修改的原因。

### 3. 统一代码（Unicode）

由于现今人类使用了接近 6800 种不同的语言，所以即使是扩展 ASCII 码这样的 8 位代码也不能满足需要。解决问题的最佳方案是设计一种全新的编码方法，这种方法必须有足够的能力来表示 6800 种语言中任意一种语言里使用的所有符号，这就是统一代码（Unicode）。

Unicode 的基本方法是用一个 16 位的数来表示 Unicode 中的每个符号，这意味着允许表示 65536 个不同的字符或符号。这种符号集被称为基本多语言平面（BMP）。这个空间已经非常大了，但设计者考虑到将来某一天它可能会不够用，所以采用了一种可使这种表示法使用得更远的方法。当用 2 个字节来表示 Unicode 字符时，使用的是 UCS-2 编码，但尽管如此，也允许在 UCS-2 文本中插入一些 UCS-4 字符。为此，在 BMP 中，保留了两个大小为 1024 的块，这两个块中，任何位置都不能用来表示任何符号。UCS-4 的两个 16 位字每个表示一个数，这个数是 UCS-2BMP 中 1024 个数值中的一个。这两个数的组合可以

表示多达 100 多万万个自定义的 UCS-4 字符。图 1-48 给出了在 PC 机中用扩展 ASCII 码、Unicode UCS-2 和 Unicode UCS-4 方法表示一个之间的差异。

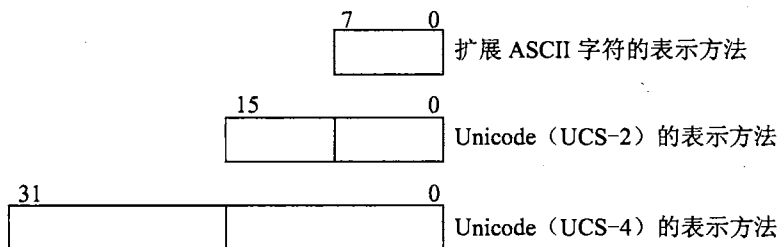


图 1-48 PC 机中表示字符的 3 种方法

随着计算机存储量的不断加大，国内计算机今后将逐步走向大字符级集国际标准 ISO/IEC10646。ISO 10646 的 BMP 与 Unicode 字符集内容相同。

#### 4. 语音编码

语音信号是模拟信号，而计算机中的信息是用二进制表示的，语音的编解码就是将语音的模拟信号转换为二进制数字信号在计算机中处理、传输，到了接收端，再将数字信号还原为模拟语音。在语音编码方面，需要了解 and 解决的问题有：

##### 1) 音调、音强和音色

声音是通过声波改变空气的疏密度，引起鼓膜振动而作用于人的听觉的。从听觉的角度，音调、音强和音色称为声音的三要素。

- 音调决定于声波的频率，声波的频率高，声音的音调就高；声波的频率低，声音的音调就低。人的听觉范围为 20Hz~20kHz。
- 音强又称响度，决定于声波的振幅。声波的振幅大，则声音强，声波的振幅小，声音弱。
- 音色决定于声波的形状。混入音波基波中的泛音不同，得到不同的音色。

##### 2) 波形采样量化

任何用符号表示的数字都是不连续的。波形的数字化过程是将连续的波形用离散的（不连续的）点近似代替的过程。在原波形上取点，称为采样。用一定的标尺确定各采样点的值（样本），称为量化。量化之后，很容易就将它们转换为二进制码。

##### 3) 采样量化的技术参数

一个数字声音的质量，决定于以下技术参数：

###### (1) 采样频率

采样频率，即一秒内的采样次数，它反映了采样点之间的间隔大小。间隔越小，丢失



的信息越少, 数字声音就越逼真细腻, 要求的存储量也就越大。由于计算机的工作速度和存储容量有限, 而且人耳的听觉上限为 20kHz, 所以采样频率不可能也不需要太高。根据奈奎斯特采样定律, 只要采样频率高于信号中的最高频率的两倍, 就可以从采样中恢复原始的波形。因此, 40kHz 以上的采样频率足以使人满意。目前多媒体计算机的采样频率有 3 个: 44.1kHz、22.05kHz 和 11.025kHz。CD 唱片采用的是 44.1kHz。

### (2) 测量精度

测量精度是样本在纵向方向的精度, 是样本的量化等级, 它通过对波形纵向方向的等分而实现。由于数字化最终要用二进制表示, 所以常用二进制数的位数表示样本的量化等级。若每个样本用 8 位二进制数表示, 则共有 8 个量级。若每个样本用 16 位二进制数表示, 则共有 16 个量级。量级越多, 采样越接近原始波形; 数字声音质量越高, 要求的存储也越大。目前多媒体计算机的标准采样量级有 8 位和 16 位两种。

### (3) 声道数

声音记录只产生一个波形, 称为单声道。声音记录只产生两个波形, 称为立体声双道。立体声比单声道声音丰满、空间感强, 但需两倍的存储空间。

## 1.4.4 差错控制编码

在数据通信方面, 嵌入式系统与通用型计算机系统一样, 要求信息传输具有高度的可靠性, 即要求误码率足够低。然而, 不管是模拟通信系统还是数字通信系统, 都存在干扰和信道传输特性不好对信号造成的不良影响, 致使数据信号在传输过程中不可避免地会发生差错。为满足通信要求, 尽可能减少和避免误码出现, 一方面要提高硬件的质量, 另一方面可以采用抗干扰编码(或差错控制编码)。

差错控制编码的基本思想是: 在发送端被传送的信息码序列的基础上, 按照一定的规则加入若干“监督码元”后进行传输, 这些加入的码元与原来的信息码序列之间存在着某种确定的约束关系。在接收数据时, 检验信息码元与监督码元之间既定的约束关系, 如该关系遭到破坏, 则在接受端可以发现传输中的错误, 乃至纠正错误。当然, 用纠(检)错来控制差错的方法来提高数据通信的可靠性是用信息量的冗余和降低系统的效率为代价来换取的。

### 1. 差错控制编码的分类

差错控制码从不同的角度出发, 有不同的分类方法。

根据码组的功能, 可分为检错码和纠错码两类。一般地说, 检错码是指能自动发现差错的码。纠错码是指不仅能发现差错而且能自动纠正差错的码。

按照信息码元与监督码元的约束关系, 又可分为分组码和卷积码两类。

按码组中监督码元与信息码元之间的关系分, 有线性码和非线性码两类。

## 2. 几种常用的差错控制编码

下面介绍几种常用的差错控制编码，它们都属于分组码一类，而且是行之有效的。

### 1) 奇偶校验码

这是一种简单的检错码，其编码规则是先将所要传输的数据码元分组，在分组数据后面附加一位监督位，使该组码连同监督位在码组中的“1”的个数为偶数（称为偶校验）或奇数（称为奇校验），在接收端按同样的规律检查，如发现不符就说明产生了差错，但是不能确定差错的具体位置，即不能纠错。

奇偶校验码的这种监督关系可以用公式表示。设码组长度为  $n$ ，表示为  $(a_{n-1}, a_{n-2}, a_{n-3}, \dots, a_0)$ ，其中前  $n-1$  位为信息码元，第  $n$  位为监督位  $a_0$ 。

在偶校验时有

$$a_0 \oplus a_1 \oplus \dots \oplus a_{n-1} = 0$$

其中  $\oplus$  表示模 2 和，监督位  $a_0$  可由下式产生

$$a_0 = a_1 \oplus a_2 \oplus \dots \oplus a_{n-1}$$

在奇校验时有

$$a_0 \oplus a_1 \oplus \dots \oplus a_{n-1} = 1$$

监督位  $a_0$  可由下式产生

$$a_0 = a_1 \oplus a_2 \oplus \dots \oplus a_{n-1} \oplus 1$$

这种奇偶校验只能发现单个或奇数个错误，而不能检测出偶数个错误，因而它的检错能力不强。

如：信息码 1110011000 按照偶监督规则插入监督位应为

$$a_0 = 1 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

信息位	监督位
1110011000	1

### 2) 海明码

海明码是 1950 年由美国贝尔实验室提出来的，是一种多重（复式）奇偶检错系统。它将信息用逻辑形式编码，以便能够检错和纠错。用在海明码中的全部传输码字是由原来的信息和附加的奇偶校验位组成的。

#### (1) 校验位的位数

在前面讨论奇偶校验时，如按偶监督，由于使用了一位监督位  $a_0$ ，故它能和信息位  $a_{n-1}, a_{n-2}, a_{n-3}, \dots, a_1$  一起构成一个代数式， $a_0 \oplus a_1 \oplus \dots \oplus a_{n-1} = 0$ 。在接收端解码时，实际上就是在计算

$$S = a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$$

当  $S=0$ , 就认为无错; 若  $S=2$ , 就认为有错。上式称为监督关系式,  $S$  称为校正子。由于校正子  $S$  的取值只有这两种, 它就只能代表有错和无错这两种信息, 而不能指出错码的位置。可以推出, 如果监督位增加一位, 即变成两位, 则能增加一个类似上式的监督关系式。两个校正子有 4 种可能的组合: 00, 01, 10, 11, 能表示 4 种不同信息。若用一种表示无错, 则其余 3 种就有可能用来表示一位错码的 3 种不同位置。同理,  $r$  个监督关系式能指示一位错码的  $(2^r-1)$  个可能位置。

一般说来, 若信息长为  $n$ , 信息位数为  $k$ , 则监督位数  $r=n-k$ 。如果希望用  $r$  个监督位构造出  $r$  个监督关系式来指示一位错码的  $n$  种可能位置, 则要求

$$2^r - 1 \geq n \text{ 或 } 2^r \geq k + r + 1$$

推导并使用信息长度为  $k$  位的码字的海明码, 所需步骤如下:

① 确定最小的校验位数  $r$ , 将它们记成  $P_1, P_2, \dots, P_r$ , 每个校验位符合不同的奇偶测试规定。

② 原有信息和  $r$  个校验位一起编成长为  $k+r$  位的新码字。选择  $r$  校验位 (0 或 1) 以满足必要的奇偶条件。

③ 对所接收的信息作所需的  $r$  个奇偶检查。

④ 如果所有的奇偶检查结果均为正确的, 则认为信息无错误。

如果发现有一个或多个错误, 则错误的位由这些检查的结果来唯一地确定。

### (2) 码字格式

从理论上讲, 校验位可放在任何位置, 但习惯上校验位被安排在 1、2、4、8、... 的位置上。当  $k=4, r=3$  时, 信息位和校验位的分布情况如表 1-13 所示 (其中,  $P_1, P_2, P_3$  为校验位,  $D_1, \dots, D_4$  为信息位)。

表 1-13 海明码中校验位和信息位的定位

码字位置	$B_1$	$B_2$	$B_3$	$B_4$	$B_5$	$B_6$	$B_7$
校验位	$x$	$x$		$x$			
信息位			$x$		$x$	$x$	$x$
复合码字	$P_1$	$P_2$	$D_1$	$P_3$	$D_2$	$D_3$	$D_4$

### (3) 校验位的确定

$r$  个校验位是通过将  $k+r$  位复合码字进行奇偶校验而确定的。其中:

- $P_1$  负责校验海明码的第 1、3、5、7、... ( $P_1, D_1, D_2, D_4, \dots$ ) 位, (包括  $P_1$  自己)
- $P_2$  负责校验海明码的第 2、3、6、7、... ( $P_2, D_1, D_3, D_4, \dots$ ) 位, (包括  $P_2$  自己)

- $P_3$  负责校验海明码的第 4、5、6、7、… ( $P_3$ 、 $D_2$ 、 $D_3$ 、 $D_4$ 、…) 位, (包括  $P_3$  自己)

如对  $k=4$ ,  $r=3$  的码字进行海明码编码, 校验位采用偶校验, 则需  $r=3$  次偶检查。这里 3 次检查分别以 ( $R_1$ 、 $R_2$ 、 $R_3$  表示) 在表 1-14 所示各位的位置上进行。

表 1-14 校验位测试

测试条件	码 字 位 置						
	1	2	3	4	5	6	7
$R_1$	x		x		x		x
$R_2$		x	x			x	x
$R_3$				x	x	x	x

可得到三个校验方程及确定校验位的三个公式 ( $B_1, B_2, \dots, B_7$  表示码字):

$$R_1 = B_1 \oplus B_3 \oplus B_5 \oplus B_7 = 0 \text{ 得 } P_1 = D_1 \oplus D_2 \oplus D_4$$

$$R_2 = B_2 \oplus B_3 \oplus B_6 \oplus B_7 = 0 \text{ 得 } P_2 = D_1 \oplus D_3 \oplus D_4$$

$$R_3 = B_4 \oplus B_5 \oplus B_6 \oplus B_7 = 0 \text{ 得 } P_3 = D_2 \oplus D_3 \oplus D_4$$

若有四位信息码 1011, 求 3 个校验位  $P_1$ 、 $P_2$ 、 $P_3$  值并生成海明码编码, 则可用上面 3 个公式解出 (如表 1-15 所示)。

表 1-15 四位信息码的海明编码

码字位置	$B_1$	$B_2$	$B_3$	$B_4$	$B_5$	$B_6$	$B_7$
码位类型	$P_1$	$P_2$	$D_1$	$P_3$	$D_2$	$D_3$	$D_4$
信息码	-	-	1	-	0	1	1
校验位	0	1	-	0	-	-	-
编码后的海明码	0	1	1	0	0	1	1

以上是发送方的处理过程, 在接收方, 也可根据这 3 个校验方程对接收到的信息进行同样的奇偶测试:

$$A = B_1 \oplus B_3 \oplus B_5 \oplus B_7 = 0;$$

$$B = B_2 \oplus B_3 \oplus B_6 \oplus B_7 = 0;$$

$$C = B_4 \oplus B_5 \oplus B_6 \oplus B_7 = 0.$$

若 3 个校验方程都成立, 即方程式右边都等于 0, 则说明没有错。若不成立即方程式右边不等于 0, 说明有错。从 3 个方程式右边的值, 可以判断哪一位出错。例如, 如果第 3 位数字反了, 则  $C=0$  (此方程没有  $B_3$ ),  $A=B=1$  (这两个方程有  $B_3$ )。可构成二进制数  $CBA$ , 以  $A$  为最低有效位, 则错误位置就可简单地用二进制数  $CBA=011$  指出。

同理,若3个方程式右边的值为001,说明第1位出错。若3个方程式右边的值为100,说明第4位出错。

### 3) 循环冗余校验码

循环冗余校验码(Cyclic Redundancy Check, CRC)是一种能力非常强的检错、纠错码,并且实现编码和检码的电路比较简单,常用于串行传送的辅助存储器与主机的数据通信和计算机网络中。

CRC的基本原理是:在 $k$ 位信息码后再拼接 $r$ 位的校验码,整个编码长度为 $n$ 位,因此,这种编码又叫 $(n, k)$ 码。对于一个给定的 $(n, k)$ 码,可以证明存在一个最高次幂为 $n-k=r$ 的多项式 $g(x)$ ,根据 $g(x)$ 可以生成 $k$ 位信息的校验码,而 $g(x)$ 叫做这个CRC码的生成多项式。

#### (1) 几个基本概念

##### ① 多项式与二进制数码

为了便于用代数理论来研究CRC码,把长为 $n$ 的码组(二进制数)与 $n-1$ 次多项式建立一一对应的关系,即把二进制数的各位当作是一个多项式的系数,二进制数的最高位对应 $x$ 的最高幂次,以下各位对应多项式的各幂次,有此幂次项对应1,无此幂次项对应0。

多项式包括生成多项式 $g(x)$ 和信息多项式 $C(x)$ 。

如多项式为 $A(x) = x^5 + x^4 + x^2 + 1$ ,可转换为二进制码组110101。

##### ② 生成多项式

生成多项式 $g(x)$ 是接收端和发送端的一个约定,也对应一个二进制数,在整个传输过程中,这个数始终保持不变。

生成多项式应满足以下条件:

- 生成多项式的最高位和最低位必须为1。
- 当被传送信息(CRC码)任何一位发生错误时,被生成多项式做模2除后应该使余数不为0。
- 不同位发生错误时,应该使余数不同。
- 对余数继续做模2除,应使余数循环。

常用的生成多项式:

$$\text{CRC}(12\text{位}) = x^{12} + x^{11} + x^3 + x^2 + x + 1$$

$$\text{CRC}(16\text{位}) = x^{16} + x^{15} + x^2 + 1$$

$$\text{CRC}(\text{CCITT}) = x^{16} + x^{12} + x^5 + 1$$

##### ③ 模2除

模2除与算术除法类似,但每一位除(减)的结果不影响其他位,即不向上一位借位。所以实际上就是异或。然后再移位做下一位的模2减。步骤如下:

首先, 用除数对被除数最高几位做模 2 减, 没有借位。然后, 除数右移一位, 若余数最高位为 1, 商为 1, 并对余数做模 2 减。若余数最高位为 0, 商为 0, 除数继续右移一位。一直做到余数的位数小于除数时, 该余数就是最终余数。

## (2) FCS 帧检验列

将信息位后添加的  $r$  位校验码, 称为信息的 FCS 帧检验列 (Frame Check Sequence)。FCS 帧检验列可由下列方法求得: 假设发送信息用信息多项式  $C(x)$  表示, 将  $C(x)$  左移  $r$  位, 则可表示成  $C(x) \times 2^r$ , 这样  $C(x)$  的右边就会空出  $r$  位, 这就是校验码的位置。通过  $C(x) \times 2^r$  除以生成多项式  $g(x)$  得到的余数就是校验码。

例如: 求信息码为 11100110, 采用 CRC 进行差错检测, 如用的生成多项式为 11001, 求 FCS 的产生过程。

解: 此题生成多项式 11001 有 5 位 ( $r+1$ ), 因此要把原始报文  $C(x)$  即 11100110 左移 4( $r$ )位变成 111001100000, 加 4 个“0”于信息尾, 就等于信息码乘以  $2^4$ , 然后被生成多项式模 2 除。

$$\begin{array}{r}
 \begin{array}{cc} 1011 & 0110 \\ 11001 \overline{) 11100110 & 0000} \\ \underline{11001} & \\ 10111 & \\ \underline{11001} & \\ 11100 & \\ \underline{11001} & \\ 101 & 00 \\ \underline{110} & 01 \\ 11 & 010 \\ \underline{11} & 001 \\ 0110 & \longrightarrow \text{FCS} \end{array}
 \end{array}$$

所以 4 位余数 0110 即为所求的 FCS

## (3) CRC 码的编码方法

编码是在已知信息位的条件下求得循环码的码组, 码组前  $k$  位为信息位, 后  $r=n-k$  位是监督位。因此, 要首先根据给定的  $(n, k)$  值选定生成多项式  $g(x)$ , 即选定一个  $r(n-k)$  次多项式作为  $g(x)$ 。因为 CRC 的理论很复杂, 本书主要介绍已有生成多项式后计算校验码的方法。结合前面 FCS 帧检验列的产生过程, 可得 CRC 码的生成步骤如下:

- ① 将  $x$  的最高幂次为  $r$  的生成多项式  $g(x)$  转换成对应的  $r+1$  位二进制数。
- ② 将信息码左移  $r$  位, 相当于对应的信息多项式  $C(x) \times 2^r$

③ 用生成多项式(二进制数)对信息码做模 2 除,得到  $r$  位的余数——FCS 帧校验列。

④ 将余数拼到信息码左移后空出的位置,得到完整的 CRC 码。

例:如上题所举例,信息 11100110 的 CRC 码为 11001100110

#### (4) CRC 码的解码方法

在发送方,CRC 码对应的发送码组多项式  $A(x)$  应能被生成多项式  $g(x)$  整除,所以在接收端可以将接收到的编码多项式  $R(x)$  用原生成多项式  $g(x)$  去做模 2 除。当传输中未发生错误时,接收码组与发送码组相同,即  $R(x) = A(x)$ ,故接收码组  $R(x)$  必定能被  $g(x)$  整除;若码组在传输中发生错误,则  $R(x) \neq A(x)$ , $R(x)$  被  $g(x)$  除时可能除不尽而有余项。因此,就以余项是否为 0 来判别码组中是否有错码。

#### (5) CRC 码的检错纠错原理

若如果有一位出错,则余数不为 0,而且不同位出错,其余数也不同。如果循环码有一位出错,用  $G(x)$  作模 2 除将得到一个不为 0 的余数。如果对余数补 0 继续除下去,将发现一个有趣的结果:各次余数将按表 1-16 顺序循环。例如第一位出错,余数将为 001,补 0 后再除(补 0 后若最高位为 1,则用除数做模 2 减取余;若最高位为 0,则其最低 3 位就是余数),得到第二次余数为 010。以后继续补 0 作模 2 除,依次得到余数为 100,011...,反复循环,这就是“循环码”名称的由来。这是一个有价值的点。如果我们在求出余数不为 0 后,一边对余数补 0 继续做模 2 除,同时让被检测的校验码字循环左移。表 1-16 说明,当出现余数(101)时,出错位也移到  $A_7$  位置。可通过异或门将它纠正后在下次移位时送回  $A_1$ 。

表 1-16 (7, 4) CRC 码的出错模式 ( $G(x) = 1011$ )

码位	收到的 CRC 码字							余数	出错位
	$A_7$	$A_6$	$A_5$	$A_4$	$A_3$	$A_2$	$A_1$		
正确	1	0	1	0	0	1	1	000	无
错误	1	0	1	0	0	1	0	001	1
	1	0	1	0	0	0	1	010	2
	1	0	1	0	1	1	1	100	3
	1	0	1	1	0	1	1	011	4
	1	0	0	0	0	1	1	110	5
	1	1	1	0	0	1	1	111	6
	0	0	1	0	0	1	1	101	7

例: CRC 码 ( $g(x) = 1011$ ), 若接收端收到的码字为 1010111, 用  $g(x) = 1011$  做模 2 除

得到一个不为 0 的余数 100, 说明传输有错。将此余数继续补 0 用  $g(x) = 1011$  作模 2 除, 同时让码字循环左移 1010111。做了 4 次后, 得到余数为 101, 这时码字也循环左移 4 位, 变成 1111010。说明出错位已移到最高位  $A_7$ , 将最高位 1 取反后变成 0111010。再将它循环左移 3 位, 补足 7 次, 出错位回到  $A_3$  位, 就成为一个正确的码字 1010011。

## 1.5 嵌入式系统的性能评价

嵌入式系统的性能评价是指为了一定目的, 按照一定的步骤, 选用一定的度量项目, 通过建模和试验, 对嵌入式系统的性能进行测试并对测试结构作出解释的技术。

嵌入式系统性能评价没有统一的规范。进行评价可以是为了不同的目的, 由不同的人员从事, 可能采用不同的度量项目, 不同的测试方法和测试工具, 对测试结果作出不同的解释。

由于嵌入式系统的复杂性, 其性能难以用几个简单的指标来描述。下面介绍主要的评价内容和方法。

### 1.5.1 度量项目

嵌入式系统性能可以从下面几个方面进行度量。

#### (1) 性能指标

分为部件性能指标和综合性能指标。嵌入式系统的硬件和软件有许多具体指标, 如加法时间、字长、存储器容量、存取时间、编译速度等。部件性能指标可以部分反映系统的性能, 但要比完整地反映一个系统的性能还要涉及到各部件的各种组织连接情况, 以及操作系统对它们的管理等。因此, 还需要综合性的指标, 有 3 种类型的综合性指标。

- 吞吐率: 如嵌入式系统在单位时间内能够处理的作业数或事务数。
- 实时性: 即对各种事件的响应时间, 如嵌入式系统获得输入到给出响应输出之间的时间。
- 各种利用率: 系统的各种资源的利用率, 即在给定的时间区间中被使用的时间与整个时间之比。

这 3 类综合性能指标都是量化的指标, 是性能评价需要重点研究的对象。

#### (2) 可靠性和安全性

高可靠性和安全性是嵌入式系统的又一大特点。由于有些嵌入式系统工作于工矿企业的现场或用于军事设备, 一旦出现故障, 就有可能造成整个生产过程混乱, 甚至产生更加严重的后果。因此, 可靠性是嵌入式系统最重要、最突出的基本要求。可靠性是一个嵌入式系统能正常工作的能力, 一般用平均故障间隔时间 MTBF 来度量。



### (3) 可维护性

可维护性是指系统失效后在固定时间内可修复到规定功能的能力。它用系统发生一次失效后,使系统返回到正常状态所需的时间来度量,它包含诊断、失效定位、失效校正等时间。一般用平均修复时间 MTTR 表示。

### (4) 可用性

可维修系统在某一时刻能提供有效使用的程度。它表示需要使用嵌入式系统时,系统所能提供使用的程度,系统的人机接口界面(GUI)的友好和易用程度。只要包括 GUI 的易用程度、系统的使用方便程度以及系统的稳定性程度等。有时也指系统实际可用时间与计划提供使用时间的比率。

### (5) 功耗

系统电能消耗。嵌入式系统对功耗具有严格的要求,从 CPU 内部的电源模式管理到外围接口,都尽量降低系统的功耗。

### (6) 环境适应性

嵌入式系统的环境适应能力是极为重要的。这是由大多数嵌入式系统所处的恶劣的工作环境所决定的,嵌入式系统必须适应用户环境的要求,这样才能保证系统长期、稳定地工作。

### (7) 通用性

嵌入式系统都是专用系统,例如:用于电力计费的 MCS-51 单片机系统与用于家庭安全监控的 MCS-51 系统,尽管在功能上有着很大的差异,但在内部结构上却有许多相似之处。因此,当某一个系统设计投入使用后,以后设计类似系统时,就可以在前一个系统的基础上添加或减少某些部件构成新的系统。

### (8) 安全性

安全性是指嵌入式系统中的程序和数据等信息的安全程度,如数据库中的数据不被破坏和不被非法修改等。

### (9) 保密性

确保系统内信息不让非法人员存取,在系统内设置的保密措施,如使用保密锁、保密码等,使个人或组织有保护和他们的数据的专门权利。

### (10) 可扩展性

指嵌入式系统的软、硬件扩充能力,以提高系统性能,如有扩展插槽,可增加插件板到系统上,又如操作系统能支持系统添加处理器、内存及其他资源等。

在上述度量项目中“环境适应性”之前的项目都有定量指标,“通用性”之后的项目是定性的。度量项目的重要程度是与用户的需求有关的,例如军用、商用或民用,会有不同的排列顺序。

上述有关处理能力的性能指标,如吞吐率、实时性和资源利用率等都与嵌入式系统工作负载有关,即与系统的输入,用户的程序、数据、命令等有关。

除性能指标外,还必须考虑嵌入式系统的开发周期、性价比。由于嵌入式系统的迅猛发展,要求嵌入式系统的设计开发周期尽可能短,争取尽快上市,占领市场;嵌入式系统对成本控制有严格的要求,尽量降低系统的成本,使系统具有尽可能高的性价比。性价比中的价格,除了直接购买嵌入式系统的价格外,还应包含安装费用、若干年的运行维修费用和软件租用费。

### 1.5.2 评价方法

评价方法大致可以分为两类:测量法和模型法。

#### 1. 测量法

通过一定的测量设备或测量程序,测得实际运行的嵌入式系统的各种性能指标或与之有关的量,然后从它们经过一些运算求出响应的性能指标,这是最直接最基本的方法。使用测量方法,必须解决以下问题:

(1) 根据研究的目的,确定要测量的系统参数。

(2) 选择测量的方式和工具。

测量的方式有两种,一种为采样方式,即每隔一定的时间间隔,对嵌入式系统的一些参数进行一次测量;另一种为事件跟踪方式,先规定一些要测量的时间,如一个作业开始执行,某个寄存器具有某种模式等,以后每当嵌入式系统出现这种事件时,就进行一次测量。

常用的测量工具分为硬件测量工具、软件测量工具、固件测量工具及混合型测量工具。目前许多测量工具已成为定型产品,可以直接购买和使用,有不少软件测量工具也已结合到系统软件中。

(3) 测量时工作负载的选择。

为了使测量所得的结果有代表性,测量时,嵌入式系统应在测量者需求的工作负载情况下运行。为此,有两种方法:一种是让嵌入式系统在日常的使用状态下运行,但选择某些与测量者要求相接近的时间区间。例如要测量系统在重负载条件下的性能,就应选择每天系统使用最忙碌的期间进行测量。另一种是由测量者编写一组能反映他们要求的典型程序,或者选择市场上已有的一些适合他们要求的典型程序,例如对于字处理和文件系统、数据库处理、图像处理、科学和工程计算等都有一些已编制好的典型程序,可供用户挑选。这些典型程序通常被称为基准程序(benchmark)。两种工作负载相比,前一种工作负载常在作系统性能监测时使用;后一种工作负载常在比较各种系统或选购新系统时使用。常见的一些嵌入式系统,性能指标大都是用某种基准程序测量的结果。



## 2. 模型法

首先对要评价的嵌入式系统建立一个适当的模型，然后求出模型的性能指标，以便对系统进行评价。此法既可用于已有的系统，也可用于尚不存在的系统，可以比较方便地应用与设计和改进。模型法与测量法是相互联系的，在模型中使用的一些参数，往往来源于对实际系统的测量结构。

模型法又分为分析模型法和模拟模型法两类。

分析模型法中使用最多的是排队模型。排队模型包括 3 个部分：

- 输入流，指各种类型的“顾客”按什么样的规则到来。
- 排队规则，指到来的顾客按怎样的规则次序接受服务，例如是先到先服务，还是按顾客的急迫程度服务。
- 服务机构，指同一时刻有多少服务设备可接纳顾客，为每一顾客需要服务多少时间。一般，服务机构过小，不能满足顾客的需求，使服务质量降低；服务机构过大，人力物力的开支增加，因此产生了顾客需求和服务机构之间的协调问题。怎样才能做到既满足顾客的需求，又使服务机构的费用最低，是排队论要研究解决的问题。在嵌入式系统中，把需要处理的各种作业、命令等当作“顾客”，把嵌入式系统的各种软、硬部件如 CPU、存储器、输入输出设备、编译模块等当作“服务员”，当某作业在 CPU 中处理时，就意味着作业（顾客）在接受这个“服务员”的服务。至于排队规则，也可以是先到先服务或有某些优先级的服务。这样就可以把排队论的许多成果应用于嵌入式系统性能评价。

为了使模型的使用对系统的评价有价值，必须解决以下 3 个问题：

- 设计模型。根据对系统和工作负载的分析、测量等提出恰当的模型。一般设计出的模型只是部分的反映出系统的性能，而且是所要关心的那部分特性。
- 解模型。如果有现成的排队论结论，就可以直接使用，不然，则需突出新的解法。
- 校准和证实模型。为了使模型化研究结果可靠，其精度必须经过校准和证实，以达到可接受的程度。

模拟模型法和分析模型法不同，它不是用一些数学方程去刻画系统的模型，而是用模型程序的运行去动态的表达嵌入式系统的状态，而进行系统统计分析，得出性能指标。这种模型可以更加详细地刻画原来的嵌入式系统，并且还可以灵活地加以控制，但构造模拟模型的费用较大，每次使用时还必须进行模拟。

这种模型法需要有一个建立模型并编制模拟程序的过程，然后校准和证实模型，才能计算出所需要的性能指标。

测量法、分析模型法和模拟模型法三者得到的结论可以相互证实。

编制模拟程序可以使用通用语言如 C, PASCAL 等，也可以使用模拟语言如 GPSS,

SIMSCRIPT 等, 目前模拟语言已有了改进, 与图形学和动画技术结合, 减少了程序设计的要求。另外, 编制模拟程序大都采用面向对象的语言, 使许多模型成为可重用的。

嵌入式系统性能评价对于设计、采购、管理、使用嵌入式系统都有非常重要的意义。嵌入式系统的发展日新月异, 评价的新课题也不断产生; 软硬件技术的进步也使评价工具不断改进。模拟语言的图形界面也大大方便了使用, 并更为直观。面向对象的语言使子模型得以重用, 方便了编写程序。预计工具的改进还将会有更大的发展。

### 1.5.3 评估嵌入式系统处理器的主要指标

嵌入式领域中许多用来分析处理器性能的标准, 例如测量处理器执行一段指定程序的速度。目前, 一般消费者能够使用的测试向量非常多, 问题是如何正确选择最为接近目标应用的测试向量。换句话说, 要先明确预期最终应用程序在待选平台上的运行情况和测试目的, 然后在挑选符合要求的特定测试向量。

#### 1. MIPS 测试基准

测试方法是计算在单位时间内各类指令的平均执行条数, 即根据各种指令的使用频度和执行时间来计算。其单位是每秒百万条指令, 表示为 MIPS。MIPS 开始是定义在 VAX 11/70 小型计算机上的, 它是第一台以 MIPS 速度运行的计算机。但许多专业人士认为 MIPS 测试结果说明不了什么问题。因为指令只是性能度量空间中的一维而已, 当把它扩展到不同体系结构上时, 其工作方式完全不同。除非是用 VAX 系列的计算机进行对比, 否则 MIPS 并没什么意义。

#### 2. Dhrystone

Dhrystone 测试基准是个简单的 C 语言程序, 它可以被编译成大约 2000 条汇编代码, 并且它不使用操作系统提供的服务功能。Dhrystone 测试基准也符合古老的 VAX 系列标准。目前, Dhrystone 是市面上最普遍适用的测试向量, 但 EEMBC 验证实验室 (EEMBC Certification Labs; ECL) 的最新研究指出, Dhrystone 不仅不适于作为嵌入式系统的测试向量, 甚至在其他大多数场合下都不适合进行应用。Dhrystone 有许多漏洞, 例如: 易被非法利用、人为痕迹明显、代码长度太短、缺乏验证及标准的运行规则等。

#### 3. EEMBC

EEMBC 测试向量是现在新兴流行的被认为比 Dhrystone 和 MIPS 更具有实际价值的测试基准。不同于 Dhrystone 测试基准, EEMBC 由其技术委员会开发, 表示实际应用中能用来测量处理器能力的算法。EEMBC (Embedded Microprocessor Benchmark Consortium) 是一个非营利性组织, 致力于帮助设计人员快速有效的选择处理器。该组织到目前为止共发布了 46 个性能测试向量, 分别应用于电信、网络、消费性产品、办公室设备和汽车电子这 5 大领域。EEMBC 测试基于每秒钟算法执行的次数和编译代码大小的统计结果。因为编译

器对代码大小和执行效率会产生巨大的影响，所以每种测试必须包括足够夺得编译器信息并设置不同的优化选项。EEMBC 发展势头很好，并有可能成为嵌入式系统开发人员进行处理器和编译器性能比较的工业标准。

需要说明的是，虽然某些定量指标能够帮助评价不同的嵌入式处理器，但是一次详尽的分析比较仍然非常重要。这些需要仔细衡量的因素包括：性能分析、功耗和效率分析、开发工具支持以及价格（必须从设备和系统角度全面考虑）。

## 第 2 章 嵌入式微处理器与接口知识

### 2.1 嵌入式微处理器的结构和类型

#### 2.1.1 嵌入式微处理器的分类

嵌入式微处理器是指应用在嵌入式计算机系统微处理器。与通用计算机系统的 CPU 相比，嵌入式微处理器具有品种多、体积小、成本低、集成度高的特点。从 1971 年 Intel 公司推出第一块微处理器芯片 4004 到今天，嵌入式微处理器已经过了 30 多年的发展历史。

如图 2-1 所示，嵌入式硬件系统一般由嵌入式微处理器、存储器和输入/输出部分组成。其中嵌入式微处理器是嵌入式硬件系统的核心，通常由 3 大部分组成：控制单元、算术逻辑单元和寄存器。各部分的主要功能如下。

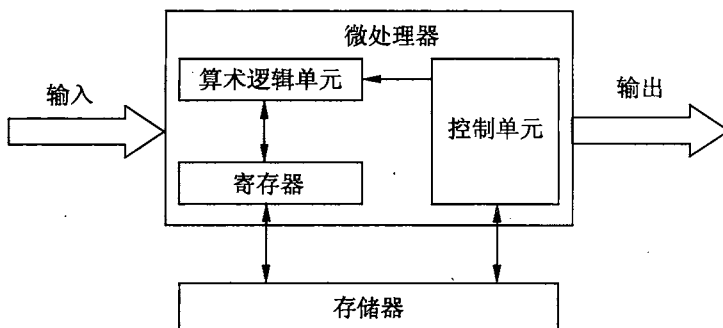


图 2-1 嵌入式硬件系统的基本结构

- 控制单元：主要负责取指、译码和取操作数等基本动作，并发送主要的控制指令。控制单元中包括两个重要的寄存器：程序计数器（PC）和指令寄存器（IR）。程序计数器用于记录下一条程序指令在内存中的位置，以便控制单元能到正确的内存位置取指；指令寄存器负责存放被控制单元所取的指令，通过译码，产生必要的控制信号送到算术逻辑单元进行相关的数据处理工作。

- 算术逻辑单元：算术逻辑单元分为两部分，一部分是算术运算单元，主要处理数值型的数据，进行数学运算，如加、减、乘、除或数值的比较；另一部分是逻辑运算单元，主要处理逻辑运算工作，如 AND、OR、XOR 或 NOT 等运算。
- 寄存器：用于存储暂时性的数据。主要是从存储器中所得到的数据（这些数据被送到算术逻辑单元中进行处理）和算术逻辑单元中处理好的数据（再进行算术逻辑运算或存入到存储器中）。

根据嵌入式微处理器的字长宽度，可分为 4 位、8 位、16 位、32 位和 64 位。一般把 16 位及以下的称为嵌入式微控制器（Embedded Micro Controller），32 位及以上的称为嵌入式微处理器。

如果按系统集成度划分，可分为两类：一种是微处理器内部仅包含单纯的中央处理器单元，称为一般用途型微处理器；另一种则是将 CPU、ROM、RAM 及 I/O 等部件集成到同一个芯片上，称为单芯片微控制器（Single Chip Microcontroller）。

如果根据用途，可分为以下几类：

（1）嵌入式微控制器（MicroController Unit, MCU），又称为单片机。微控制器的片上外设资源一般比较丰富，适合于控制，因此称为微控制器。微控制器芯片内部集成 ROM/EPROM、RAM、总线、总线逻辑、定时/计数器、看门狗、I/O、串行口、脉宽调制输出（PWM）、A/D、D/A、Flash、EEPROM 等各种必要功能和外设。和嵌入式微处理器相比，微控制器的最大特点是单片化，体积大大减小，从而使功耗和成本下降，可靠性提高。由于嵌入式微控制器低廉的价格、优良的功能，所以拥有的品种和数量最多，比较有代表性的包括 8051、MCS-251、MCS-96/196/296、C166/167、68K 系列以及 MCU 8XC930/931、C540、C541，并且有支持 I<sup>2</sup>C、CAN-BUS、LCD 及众多专用嵌入式微控制器和兼容系列。目前嵌入式微控制器占嵌入式系统约 70% 的市场份额。

（2）嵌入式微处理器。嵌入式微处理器（Embedded Micro Processing Unit, EMPU）由通用计算机中的 CPU 发展而来。它的特征是具有 32 位以上的处理器，具有较高的性能，当然其价格也相应较高。但与计算机 CPU 不同的是，在实际嵌入式应用中，只保留和嵌入式应用紧密相关的功能硬件，去除其他的冗余功能部分，这样就以最低的功耗和资源实现嵌入式应用的特殊要求。嵌入式微处理器与用于桌面计算机的 CPU 相比具有体积小、重量轻、功耗低、成本低及可靠性高的优点。通常嵌入式微处理器把 CPU、ROM、RAM 及 I/O 等元件做到同一个芯片上，也称为单板计算机。当前，主要的嵌入式微处理器有 ARM、MIPS、POWER PC 和基于 X86 的 386EX 等。

（3）嵌入式 DSP 处理器（Digital Signal Processor, DSP），是专门用于信号处理方面的处理器，其在系统结构和指令算法方面进行了特殊设计，具有很高的编译效率和指令执行速度。在数字滤波、FFT、频谱分析等各种仪器上 DSP 获得了大量的应用。

DSP 芯片内部采用程序和数据分开存储和传输的哈佛结构, 具有专门硬件乘法器, 广泛采用流水线操作, 提供特殊的 DSP 指令, 可用来快速地实现各种数字信号处理算法, 加之集成电路的优化设计, 使其处理速度比最快的 CPU 还快 10~50 倍。

DSP 发展历程大致分 3 个阶段: 20 世纪 70 年代理论先行, 20 世纪 80 年代产品普及, 20 世纪 90 年代突飞猛进。在 DSP 出现之前, 数字信号处理只能依靠微处理器 (MPU) 来完成, 但微处理器的低处理速度无法满足高速实时的要求, 因此直到 20 世纪 70 年代, 才有人提出了 DSP 的理论和算法基础。那时的 DSP 仅停留在教科书上, 即便是研制出来的 DSP 系统也是由分立元件组成的, 其应用领域仅局限于军事、航空航天部门。

随着大规模集成电路技术的发展, 1982 年, 世界上诞生了首枚 DSP 芯片。这种 DSP 器件采用微米工艺 NMOS 技术制作, 虽功耗和尺寸稍大, 但运算速度却比微处理器快了数十倍, 尤其在语音合成和编码解码器中得到了广泛应用。DSP 芯片的问世是个里程碑, 它标志着 DSP 应用系统由大型系统向小型化迈进了一大步。到 20 世纪 80 年代中期, 随着 CMOS 技术的进步与发展, 第 2 代基于 CMOS 工艺的 DSP 芯片应运而生, 其存储容量和运算速度都得到成倍提高, 成为语音处理和图像硬件处理技术的基础。

20 世纪 80 年代后期, 第 3 代芯片问世, 运算速度进一步提高, 其应用范围逐步扩大到通信和计算机领域。

20 世纪 90 年代, DSP 发展最快, 相继出现了第 4 代和第 5 代 DSP 器件。现在的 DSP 属于第 5 代产品, 与第 4 代相比, 其系统集成度更高, 并将 DSP 芯核及外围元件综合集成在单一芯片上。这种集成度极高的 DSP 芯片不仅在通信和计算机领域发挥重要作用, 而且逐渐渗透到人们日常的消费领域。

(4) 嵌入式片上系统 (System On Chip, SOC), 是追求产品系统最大包容的集成器件。SOC 最大的特点是成功实现了软硬件无缝结合, 直接在处理器片内嵌入操作系统的代码模块。而且 SOC 具有极高的综合性, 在一个硅片内部运用 VHDL 等硬件描述语言, 实现一个复杂的系统。用户不需要再像传统的系统设计一样, 绘制庞大复杂的电路板, 一点点地连接焊制, 只需要使用精确的语言, 综合时序设计直接在器件库中调用各种通用处理器的标准, 然后通过仿真之后就可以直接交付芯片厂商进行生产。由于绝大部分系统构件都是在系统内部, 整个系统就特别简洁, 不仅减小了系统的体积和功耗, 而且提高了系统的可靠性, 提高了设计生产效率。

由于 SOC 往往是专用的, 所以大部分都不为用户所知, 比较典型的 SOC 产品是 Philips 公司的 Smart XA。少数通用系列如 Siemens 公司的 TriCore、Motorola 公司的 M-Core、某些 ARM 系列器件、Echelon 和 Motorola 联合研制的 Neuron 芯片等。预计不久的将来 SOC 芯片也将在声音、图像、影视、网络及系统逻辑等应用领域中发挥重要作用。



## 2.1.2 典型 8 位微处理器的结构和特点

8 位微处理器是指使用 8 位数据总线的微处理器。大部分的 8 位微处理器有 16 位的地址总线,其能够访问 64KB 的地址空间,而 8 位的数据总线则可以通过多重内存存取的方式来处理更多的数据。最早的 8 位微处理器是 1973 年时 Intel 公司所开发的 8080 微处理器芯片,随后各大厂商也陆续推出 8 位微处理器,如 Zilog 公司的 Z80、Motorola 公司的 6800、National 半导体公司的 NSC800 及 Intel 公司的 8085 等。

由于 8 位微处理器具有低成本、可扩充内存及接口设备等,目前仍然在嵌入式系统领域得到广泛应用。8 位的微处理器有许多种不同时代的产品,其中有两个比较著名,一个是 Intel 公司推出的 8048,另一个则是 Fairchild 及 Mostek 公司推出的 3870。Intel 公司的 8048 在当时是一种新的体系结构,并未延续其他已存在的微处理器体系结构,因此在指令集及体系结构的开发上变的有些困难,但因为它是定位在具可伸缩性并且低成本的产品控制单元,所以至今仍被广泛地使用。另外,其所衍生的第二代产品 8051,更是目前应用最广泛的 8 位微处理器系列。Intel 的 8041 及 8042 是延续 8048 的系统,并作为从处理器(Slave Processor)使用。8044 是 8051 的延续微处理器,它包含了一个额外的链表接口,可以连到主微处理器,做其他的数据处理。

8051 是 8 位微处理器中的典型产品,下面将对这典型微处理器进行系统的介绍。

### 1. 硬件结构

8051 单片机包含 CPU、ROM(程序存储器)、RAM(数据存储器)、定时/计数器、并行接口、串行接口和中断系统等几大单元及数据总线、地址总线和控制总线等三大总线,下面分别加以说明。

#### (1) CPU

CPU(中央处理器)是整个单片机的核心部件,是 8 位数据宽度的处理器,能处理 8 位二进制数据或代码。CPU 负责控制、指挥和调度整个单元系统协调的工作,完成运算和控制输入输出功能等操作。

#### (2) ROM(程序存储器)

8051 共有 4096 个 8 位掩膜 ROM,用于存放用户程序、原始数据或表格。

#### (3) RAM(数据存储器)

8051 内部有 128 个 8 位用户数据存储单元和 128 个专用寄存器单元,它们是统一编址的,专用寄存器只能用于存放控制指令数据。用户只能访问,而不能用于存放用户数据,所以,用户能使用的 RAM 只有 128 个,可存放读写的数据,运算的中间结果或用户定义的字型表。

#### (4) 定时/计数器

8051 有两个 16 位的可编程定时/计数器,以实现定时或计数产生中断,用于控制程序

跳转。

#### (5) 并行输入输出 (I/O) 接口

8051 共有 4 组 8 位 I/O 接口 (P0、P1、P2 或 P3)，用于对外部数据的传输。

#### (6) 全双工串行接口

8051 内置一个全双工串行通信口，用于与其他设备间的串行数据传送。该串行接口既可以用作异步通信收发器，也可以当同步移位器使用。

#### (7) 中断系统

8051 具备较完善的中断功能，有两个外部中断、两个定时/计数器中断和一个串行中断，可满足不同的控制要求，并具有 2 级的中断优先级选择。

#### (8) 时钟电路

8051 内置时钟电路，最高频率达 12MHz，用于产生整个单片机运行的脉冲时序，但 8051 单片机需外置振荡电容。

Intel 公司的 MCS-51 系列单片机采用的是哈佛体系结构。图 2-2 所示为 MCS-51 系列单片机的内部结构示意图。

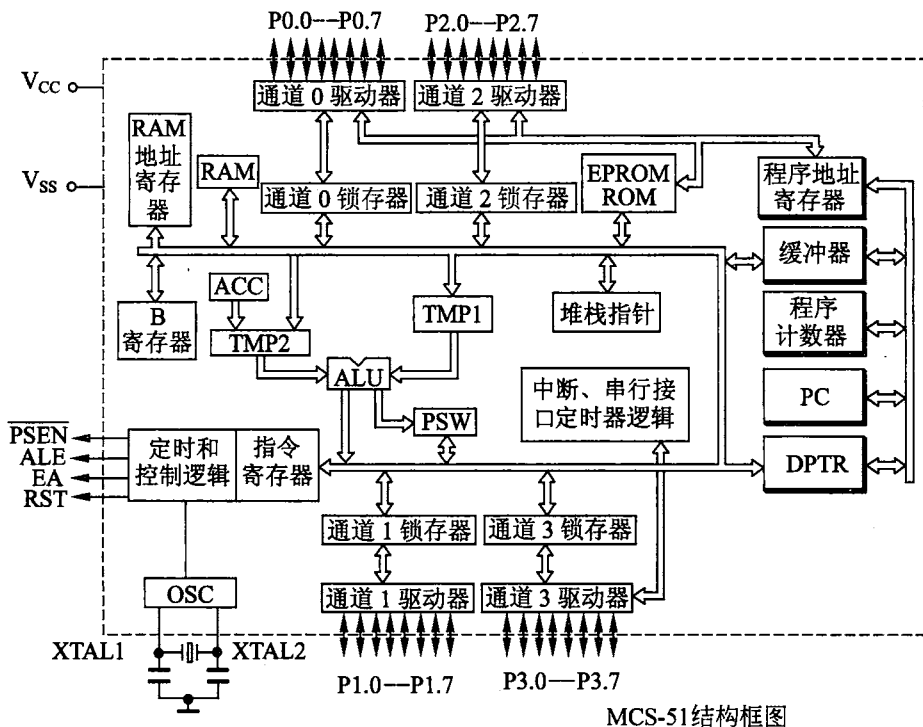


图 2-2 MCS-51 系列微处理器的内部结构示意图

### (9) MCS-51 的引脚说明

MCS-51 系列单片机中的 8031、8051 及 8751 均采用 40 引脚封装的双列直接 DTP 结构, 如图 2-3 所示, 在 40 个引脚中, 其中正电源和地线两根, 外置石英振荡器的时钟线两根, 4 组 8 位共 32 个 I/O 接口, 中断接口线与 P3 接口线复用。现在对这些引脚的功能加以说明。

- Pin20: 地引脚。
- Pin40: 正电源引脚, 正常工作或对片内 EPROM 编写程序时, 接+5V 电源。
- Pin19: 时钟 XTAL1 引脚, 片内振荡电路的输入端。
- Pin18: 时钟 XTAL2 引脚, 片内振荡电路的输出端。

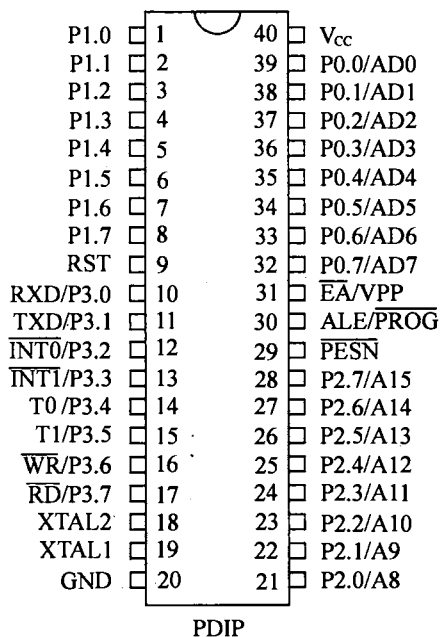


图 2-3 MCS-51 系列微处理器的引脚配置

8051 的时钟有两种方式, 如图 2-4 所示, 一种是片内时钟振荡方式, 但需在 18 和 19 引脚外接石英晶体 (2-12MHz) 和振荡电容, 振荡电容的值一般取 10p~30p。另外一种方式是外部时钟方式, 即将 XTAL1 接地, 外部时钟信号从 XTAL2 脚输入。

#### • 输入输出 (I/O) 引脚

Pin39~Pin32 为 P0.0~P0.7 输入输出脚, Pin1~Pin8 为 P1.0~P1.7 输入输出脚, Pin21~Pin28 为 P2.0~P2.7 输入输出脚, Pin10~Pin17 为 P3.0~P3.7 输入输出脚, 这些输入输出脚的功能说明将在以下内容阐述。

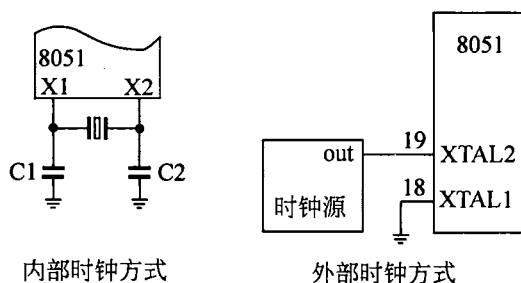


图 2-4 8051 的时钟有两种方式

- Pin9: RESET/V<sub>pd</sub> 复位信号复用脚

当 8051 通电, 时钟电路开始工作, 在 RESET 引脚上出现 24 个时钟周期以上的高电平, 系统即初始复位。初始化后, 程序计数器 PC 指向 0000H, P0~P3 输出口全部为高电平, 堆栈指针写入 07H, 其他专用寄存器被清 0。RESET 由高电平下降为低电平后, 系统即从 0000H 地址开始执行程序。然而, 初始复位不改变 RAM (包括工作寄存器 R0-R7) 的状态。

8051 的初始态如表 2-1 所示。

表 2-1 8051 的初始态

特殊功能寄存器	初始态	特殊功能寄存器	初始态
ACC	00H	B	00H
PSW	00H	SP	07H
DPH	00H	TH0	00H
DPL	00H	TL0	00H
IP	xxx00000B	TH1	00H
IE	0xx00000B	TL1	00H
TM OD	00H	TCO N	00H
SCON	xxxxxxxB	SBUF	00H
P0-P3	1111111B	PCON	0xxxxxxxB

8051 的复位方式可以是自动复位, 也可以是手动复位, 如图 2-5 所示。此外, RESET/V<sub>pd</sub> 还是一复用引脚, V<sub>CC</sub> 掉电期间, 此脚可接上备用电源, 以保证单片机内部 RAM 的数据不丢失。

- Pin30

当访问外部程序器时, ALE (地址锁存) 的输出用于锁存地址的低位字节。而访问内部程序存储器时, ALE 端将有一个 1/6 时钟频率的正脉冲信号, 这个信号可以用于识别单

片机是否工作，也可以当作一个时钟向外输出。还有一个特点，当访问外部程序存储器，ALE 会跳过一个脉冲。

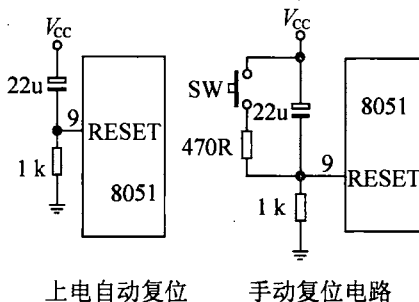


图 2-5 8051 的复位方式

#### • Pin29

当访问外部程序存储器时，此引脚输出负脉冲选通信号，PC 机的 16 位地址数据将出现在 P0 和 P2 接口上，外部程序存储器则把指令数据放到 P0 接口上，由 CPU 读入并执行。

#### • Pin31

EA/V<sub>pp</sub> 程序存储器的内外部选通线，8051 和 8751 单片机，内置有 4KB 的程序存储器，当 EA 为高电平并且程序地址小于 4KB 时，读取内部程序存储器指令数据，而地址超过 4KB 则读取外部指令数据。如 EA 为低电平，则不管地址大小，一律读取外部程序存储器指令。显然，对内部无程序存储器的 8031，EA 端必须接地。在编程时，EA/V<sub>pp</sub> 引脚还需加上 21V 的编程电压。

## 2. 指令集

MCS-51 共有 111 条指令，可分为 5 类。

### (1) 数据传送类指令

数据传送指令共有 29 条，数据传送指令一般的操作是把源操作数传送到目的操作数，指令执行完成后，源操作数不变，目的操作数等于源操作数。如果要求在进行数据传送时，目的操作数不丢失，则不能用直接传送指令，而采用交换型的数据传送指令，数据传送指令不影响标志 C、AC 和 OV，但可能会对奇偶标志 P 有影响。

数据传送类指令包含以下指令：

- 以累加器 A 为目的操作数类指令（4 条）
- 以寄存器 Rn 为目的操作数的指令（3 条）
- 以直接地址为目的操作数的指令（5 条）
- 以间接地址为目的操作数的指令（3 条）

- 查表指令（2条）
- 累加器 A 与片外数据存储器 RAM 传送指令（4条）
- 堆栈操作类指令（2条）
- 交换指令（5条）
- 16 位数据传送指令（1条）

## （2）算数运算类指令

算术运算指令共有 24 条，算术运算主要是执行加、减、乘、除法四则运算。另外 MCS-51 指令系统中有相当一部分是进行加 1、减 1 操作，BCD 码的运算和调整，都归类为运算指令。虽然 MCS-51 单片机的算术逻辑单元 ALU 仅能对 8 位无符号整数进行运算，但利用进位标志 C，则可进行多字节无符号整数的运算。同时利用溢出标志，还可以对带符号数进行补码运算。需要指出的是，除加 1、减 1 指令外，这类指令大多数都会对 PSW（程序状态字）有影响。这在使用中应特别注意。

- 加法指令（4条）
- 带进位加法指令（4条）
- 带借位减法指令（4条）
- 乘法指令（1条）
- 除法指令（1条）
- 加 1 指令（5条）
- 减 1 指令（4条）
- 十进制调整指令（1条）

## （3）逻辑运算及移位类指令

逻辑运算和移位指令共有 25 条，有与、或、异或、求反、左右移位、清 0 等逻辑操作，有直接、寄存器和寄存器间址等寻址方式。这类指令一般不影响程序状态字（PSW）标志。

- 循环移位指令（4条）
- 累加器半字节交换指令（1条）
- 求反指令（1条）
- 清零指令（1条）
- 逻辑与操作指令（6条）
- 逻辑或操作指令（6条）
- 逻辑异或操作指令（6条）

## （4）控制转移类指令

控制转移指令用于控制程序的流向，所控制的范围即为程序存储器区间，MCS-51 系



列单片机的控制转移指令相对丰富，有可对 64KB 程序空间地址单元进行访问的长调用、长转移指令，也有可对 2KB 字节进行访问的绝对调用和绝对转移指令，还有在一页范围内短相对转移及其他无条件转移指令，这些指令的执行一般都不会对标志位有影响。

- 无条件转移指令（4 条）
- 条件转移指令（8 条）
- 子程序调用指令（1 条）
- 空操作指令（1 条）

#### （5）布尔变量操作类指令

布尔处理功能是 MCS-51 系列单片机的一个重要特征，这是出于实际应用需要而设置的。布尔变量也即开关变量，它是以位（bit）为单位进行操作的。

在物理结构上，MCS-51 单片机有一个布尔处理器，它以进位标志做为累加位，以内部 RAM 可寻址的 128 个为存储位。

既然有布尔处理器功能，所以也就有相应的布尔操作指令集，下面我们分别讨论。

- 位传送指令（2 条）
- 位置位复位指令（4 条）
- 位运算指令（6 条）
- 位控制转移指令（5）

### 3. MCS-51 的寻址方式

MCS-51 的寻址方式较多，使用起来也很方便，灵活性强。下面分别介绍几种寻址方式的原理。

#### （1）直接寻址

指令中操作数直接以单元地址形式出现，例如：

**MOV A, 68H**

这条指令的意义是把内部 RAM 中的 68H 单元中的数据内容传送到累加器 A 中。值得注意的是直接寻址方式只能使用 8 位二进制地址，因此这种寻址方式仅限于内部 RAM 进行寻址。低 128 位单元在指令中直接以单元地址的形式给出。对于特殊功能寄存器可以使用其直接地址进行访问，还可以以它们的符号形式给出，只是特殊功能寄存器只能用直接寻址方式访问，而无其他方法。

#### （2）寄存器寻址

寄存器寻址对选定的 8 个工作寄存器 R0~R7 进行操作，也就是操作数在寄存器中，因此指定了寄存器就得到了操作数，寄存器寻址的指令中以寄存器的符号来表示寄存器，例如：

**MOV A, R1**



这条指令的意义是把所用的工作寄存器组中的 R3 的内容送到累加器 A 中。

值得一提的是工作状态寄存器的选择是通过程序状态字寄存器来控制的, 在这条指令前, 应通过 PSW 设定当前工作寄存器组。

### (3) 寄存器间接寻址

在寄存器寻址方式中, 寄存器中存放的是操作数。而寄存器间接寻址方式中, 寄存器中存放的则为操作数的地址, 也即操作数是通过寄存器指向的地址单元得到的, 这便是寄存器间接寻址名称的由来。

例如:

**MOV A, @R0**

这条指令的意义是把 R0 寄存器指向地址单元中的内容送到累加器 A 中。假如 R0=#56H, 就是把 56H 单元中的数据送到累加器 A 中。

寄存器间接寻址方式可用于访问内部 RAM 或外部数据存储器。访问内部 RAM 或外部数据存储器的低 256 字节时, 可通过 R0 和 R1 作为间接寄存器。然而, 内部 RAM 的高 128 字节地址与专用寄存器的地址是重叠的, 所以这种寻址方式不能用于访问特殊功能寄存器。

外部数据存储器的空间为 64KB, 这时可采用 DPTR 作为间址寄存器进行访问, 指令如下:

**MOVX A, @DPTR**

这条指令的意义是与上述类似, 不再赘述。

### (4) 立即寻址

立即寻址就是把操作数直接在指令中给出, 即操作数包含在指令中, 指令操作码的后面紧跟着操作数, 一般把指令中的操作数称为立即数, 因此而得名。为了与直接寻址方式相区别, 在立即数前加上 “#” 符号, 例如:

**MOVX A, #0EH**

这条指令的意义是将 0EH 这个操作数送到累加器 A 中。

### (5) 变址寻址

变址寻址是以 DPTR 或 PC 作为基址寄存器, 以累加器 A 作为变址寄存器, 将两寄存器的内容相加形成 16 位地址形成操作数的实际地址。例如:

**MOV A, @A+DPTR**

**MOVX A, @A+PC**

**JMP @A+DPTR**

在这 3 条指令中, A 作为偏移量寄存器, DPTR 或 PC 作为变址寄存器, A 作为无符号数与 DPTR 或 PC 的内容相加, 得到访问的实际地址。其中前两条是程序存储器读指令, 后一条是无条件转移指令。



### (6) 位寻址

在 MCS-51 单片机中, RAM 中的 20H~2FH 字节单元对应的位地址为 00H~7FH, 特殊功能寄存器中的某些位也可进行位寻址, 这些单元既可以采用字节方式访问它们, 也可采用位寻址的方式访问它们。

### (7) 相对寻址

相对寻址方式是为了程序的相对转移而设计的, 是以 PC 的内容为基址, 加上给出的偏移量作为转移地址, 从而实现程序的转移。转移的目的地址可参见如下表达式:

$$\text{目的地址} = \text{转移指令地址} + \text{转移指令字节数} + \text{偏移量}$$

值得注意的是, 偏移量是有正负号之分的, 偏移量的取值范围是当前 PC 值的-128~+127 之间。

## 2.1.3 典型 16 位微处理器的结构和特点

继 8 位的微处理器后, 许多厂商为了满足更复杂的应用, 推出了 16 位微处理器。16 位微处理器是指内部总线宽度为 16 位的微处理器。16 位微处理器的操作速度及数据吞吐能力在性能上比 8 位微处理器有较大的提高, 它的数据宽度增加了一倍, 实时处理能力更强, 主频更高, 集成度、RAM 和 ROM 都有较大的增加, 而且有更多的中断源, 同时配置了多路的 A/D 转换通道和高速的 I/O 处理单元, 适用于更复杂的控制系统。

Intel 公司的 8086 是第一款 16 位微处理器, 当时 IBM 公司推出的个人计算机都是采用 8086 作为个人计算机的数据处理及控制核心。8086 微处理器延续了 Intel 公司之前的 8080 及 8085 微处理器的基本体系结构, 再加上一些增强式的硬件体系结构与指令集。Intel 公司随后又在 1982 年 2 月, 推出了第二代的 8086 产品 80286 微处理器, 集成了以往许多微处理器需额外加上的外围设备组件, 包括: 一个时钟产生器、两个直接内存访问信道、一个中断信号控制器、3 个可程序化计时单元、可程序化芯片选择逻辑单元以及一个等待状态产生器; 并且和 8086 及 8088 微处理器的软件兼容, 因而受到市场的欢迎。

目前 16 位微控制器以 Intel 公司的 MCS-96/196 系列、TI 公司的 MSP430 系列和 Motorola 公司的 68H12 系列为主, 它们主要应用于便携式设备、工业控制及智能仪器仪表等。下面以 Motorola 公司的 16 位微处理器 MC68HC912DG128A 为例, 对其硬件结构及其组成进行介绍。

### 1. MC68HC912DG128A 的硬件结构

MC68HC912DG128A 微处理器具有丰富的指令系统、运算速度快、编程效率高。内部集成 128KB 的 Flash、8KB RAM, 且支持 C 语言, 支持后台背景调试 (Background Debug Mode, BDM) 方式, 开发调试都很方便。具有集成度高、功能模块强的优点。

MC68HC912DG128A 的模块框图如图 2-6 所示。下面对其组成进行介绍。



(1) CPU12。中央处理器 CPU12 由以下 3 部分组成：算术逻辑单元(ALU)、控制单元和寄存器组。寄存器组如图 2-6 所示。CPU 内部总线频率为 8MHz，寻址方式有 16 种，堆栈指针和变址寄存器均为 16 位。它具有很强的高级语言支持功能。CPU 的累加器 A 和 B 是 16 位的，也可以组成 32 位累加器 D。

CPU12 的寄存器组包括如下 5 个部分：

- 16 位累加器 A、B 或 32 位的累加器 D。
- 16 位变址寄存器 X 和 Y 是用来处理操作数的地址。可分别用于源地址和目的地址的指针型变量运算。
- 堆栈指针 (SP) 是 16 位寄存器。
- 程序计数器 (PC) 是 16 位寄存器，它表示下一条指令或下一个操作数的地址。
- 条件码寄存器 (CCR)。

(2) 存储器。

- 128KB FLASH 存储器。
- 8KB RAM。
- 2KB EEPROM。

(3) 多元化总线。可以工作在单片方式，也可以通过总线扩展存储空间和增加 I/O 芯片，工作在扩展方式。地址总线 16 位，数据线 16 位或 8 位，地址和数据总线占用 3 个或 4 个 8 位 I/O 并行接口，在单片方式下这 32 位可做普通 I/O 接口用。

(4) 两个 8 路 10 位 A/D 转换器。

(5) 控制器局域网模块 (CAN)。68HC912DG128A 内部有两个 CAN 模块，每个 CAN 具有两个接收缓冲区和 3 个发送缓冲区。每个 CAN 有 RX、TX、出错、唤醒 4 个独立的中断通道。

CAN 模块具有自检功能，有低通滤波唤醒功能。

(6) 增强型捕捉定时器。

- 16 位主计数器，7 位分频系数。
- 8 个输入捕捉通道或输出比较通道，其中 4 个输入捕捉通道带有缓存。
- 4 个 8 位或 2 个 16 位脉宽计数器。
- 每个信号滤波器有 4 个用户可选择的延迟计数器。

(7) 脉宽调制模块(PWM)可设置成 4 路 8 位或者 2 路 16 位，逻辑时钟选择频率宽。

(8) 串行接口。

- 两个异步串行通信接口 (SCI) 模块。
- 一个 I2C 总线接口。
- 一个同步串行外设接口 SPI。

(9) 两个具有产生中断、唤醒 CPU 功能的 8 位并行接口，也可以设为输出。

(10) 时钟发生器。

- 具有锁相环频率合成器。这是时钟发生器中的重要电路。它的存在使外部 32MHz 晶振可以产生 8MHz 的总线频率。
- 也可使用 0.5MHz~16MHz 的低功耗晶振。

(11) 开发支持。

- 支持单线背景调试模式 (BDM)。
- 支持高级语言编程。

### 2.1.4 典型 32 位微处理器的结构和特点

32 位微处理器采用 32 位的地址和数据总线，其地址空间达到了  $2^{32}=4\text{GB}$ 。目前主流的 32 位嵌入式微处理器系列主要有 ARM 系列、MIPS 系列、PowerPC 系列等。属于这些系列的嵌入式微处理器产品很多，有千种以上。

#### 1. ARM 处理器

##### (1) ARM 概述

ARM (Advanced RISC Machine) 公司是一家专门从事芯片 IP 设计与授权业务的英国公司，其产品有 ARM 内核以及外围接口。ARM 内核是一种 32 位 RISC 微处理器，具有功耗低、性价比高和代码密度高等特点。

目前，70% 的移动电话、大量的游戏机、手持 PC 和机顶盒等都已采用了 ARM 处理器，许多一流的芯片厂商都是 ARM 的授权用户，如 Intel、Samsung、TI、Freescale、ST 等公司。

ARM 微处理器体系结构目前被公认为是嵌入式应用领域领先的 32 位嵌入式 RISC 微处理器结构。自诞生至今，ARM 体系结构发展并定义了 7 种不同的版本。从版本 1 到版本 7，ARM 体系的指令集功能不断扩大。ARM 处理器系列中的各种处理器，虽然在实现技术、应用场合和性能方面都不相同，但只要支持相同的 ARM 体系版本，基于它们的应用软件是兼容的。表 2-2 给出了 ARM 体系结构各版本的特点。

表 2-2 ARM 体系结构版本及特点

版 本	ARM 处理器系列	特 点
ARMv1	ARM1	<p>该版体系结构只在原型机 ARM1 出现过，没有用于商业产品。其基本性能有：</p> <ul style="list-style-type: none"> <li>• 基本的数据处理指令（无乘法）</li> <li>• 26 位寻址</li> </ul>

版 本	ARM 处理器系列	特 点
ARMv2	ARM2 和 ARM3	<p>该版体系结构对 ARMv1 版进行了扩展, 版本 ARMv2a 是 v2 版的变种, ARM3 芯片采用了 ARMv2a。ARMv2 版增加了以下功能:</p> <ul style="list-style-type: none"> <li>• 32 位乘法和乘加指令</li> <li>• 支持 32 位协处理器操作指令</li> <li>• 快速中断模式</li> </ul>
ARMv3 ARMv3M	ARM6、ARM7DI、 ARM7M、	<p>ARMv3 版体系结构对 ARM 体系结构作了较大的改动:</p> <ul style="list-style-type: none"> <li>• 寻址空间增至 32 位 (4GB)</li> <li>• 独立的当前程序状态寄存器 CPSR 和程序状态保存寄存器 SPSR, 保存程序异常中断时的程序状态, 以便于对异常的处理</li> <li>• 增加了中止 (Abort) 和未定义两种处理器模式</li> <li>• 增加了 MMU 支持</li> <li>• ARMv3M 增加了有符号和无符号长乘法指令</li> </ul>
ARMv4 ARMv4T	StrongARM、 ARM7TDMI、 ARM9T	<p>ARMv4 版体系结构是目前应用最广的 ARM 体系结构, 在 v3 版上作了进一步扩充, 指令集中增加了以下功能:</p> <ul style="list-style-type: none"> <li>• 增加了系统模式</li> <li>• 增加了 16 位 Thumb 指令集</li> <li>• 完善了软件中断 SWI 指令的功能</li> <li>• 不再支持 26 位寻址模式</li> </ul>
ARMv5TE ARMv5TEJ	ARM9E、 ARM10E、Xscale、 ARM7EJ、 ARM926EJ	<p>ARMv5 版体系结构是在 ARMv4 版基础上增加了一些新的指令, 包括:</p> <ul style="list-style-type: none"> <li>• 增加 ARM 与 Thumb 状态之间切换的指令</li> <li>• 增强乘法指令和快速乘累加指令</li> <li>• 增加了数字信号处理指令 (ARMv5TE 版)</li> <li>• 增加了 Java 加速功能 (ARMv5TEJ 版)</li> </ul>
ARMv6	ARM11	<p>ARMv6 版体系结构是 2001 年发布的, 首先在 2002 年春季发布的 ARM11 处理器中使用。此体系结构在 ARMv5 版基础上增加了以下功能:</p> <ul style="list-style-type: none"> <li>• Thumb-2 增强代码密度</li> <li>• SIMD 增强媒体和数字处理功能</li> <li>• TrustZone 提供增强的安全性能</li> <li>• IEM 提供增强的功耗管理功能</li> </ul>
ARMv7	Cortex 系列	<p>ARMv7 版体系结构定义了 3 种不同的微处理器系列:</p> <ul style="list-style-type: none"> <li>• A 系列 面向应用的微处理器核, 支持复杂操作系统和用户应用</li> <li>• R 系列 深度嵌入的微处理器核, 针对实时系统应用</li> <li>• M 系列 微控制核, 针对成本敏感的嵌入式控制应用</li> </ul>

作为一种 RISC 体系结构的微处理器, ARM 处理器具有 RISC 体系结构的典型特征, 同时具有以下特点:

- 在每条数据处理指令当中, 都控制算术逻辑单元 ALU 和移位器, 以使 ALU 和移位器获得最大的利用率。
- 自动递增和自动寻址模式, 以优化程序中的循环。
- 同时执行 Load 和 Store 多条指令, 以增加数据吞吐量。
- 所有指令都可以条件执行, 以执行吞吐量。

这些是对基本 RISC 体系结构的增强, 使得 ARM 处理器可以在高性能、小代码尺寸、低功耗和小芯片面积之间获得好的平衡。

### (2) ARM 的数据类型

- 字 (Word): 在 ARM 体系结构中, 字的长度为 32 位, 而在 8 位/16 位处理器体系结构中, 字的长度一般为 16 位。
- 半字 (Half-Word): 在 ARM 体系结构中, 半字的长度为 16 位, 与 8 位/16 位处理器体系结构中字的长度一致。
- 字节 (Byte): 在 ARM 体系结构和 8 位/16 位处理器体系结构中, 字节的长度均为 8 位。

### (3) ARM 的运行模式

ARM 处理器有 7 种运行模式, 如表 2-3 所示。大多数应用程序在 User 模式下执行, 当特定的异常出现时, 进入相应的 6 种异常模式之一。每种模式都有某些附加的寄存器保存相应的状态。除 User 模式外, 其他模式都被称为特权模式, 可以存取系统中的任何资源。User 模式下程序不能访问有些受保护的资源, 也不能直接改变 CPU 的模式, 而只能通过异常的形式来改变 CPU 的当前运行模式。软件可以控制 CPU 模式的改变, 外部中断也可以引起模式的改变。

表 2-3 ARM 处理器的 7 种运行模式

处理器模式	说 明
用户模式 (User)	正常程序执行模式, 用于应用程序
异常模式 (FIQ)	快速中断处理, 用于支持高速数据传送通道处理
异常模式 (IRQ)	用于一般中断处理
异常模式 (Supervisor)	特权模式, 用于操作系统
异常模式 (Abort)	存储器保护异常处理
异常模式 (Undefined)	未定义指令异常处理
系统模式 (System)	运行特权操作系统任务 (ARM V4 以上版本)

#### (4) 寄存器结构

如图 2-7 所示, ARM 微处理器共有 37 个 32 位寄存器, 其中 31 个为通用寄存器, 6 个为状态寄存器。但是这些寄存器不能被同时访问, 具体哪些寄存器是可编程访问的, 取决于微处理器的工作状态及具体的运行模式。但在任何时候, 通用寄存器 R14~R0、程序计数器 PC、一个或两个状态寄存器都是可访问的。

ARM 状态下的通用寄存器与程序计数器

System & User	FIQ	Supervisor	About	IRG	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

ARM 状态下的程序状态寄存器

CPSR	CPSR SPSR_fiq	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_irq	CPSR SPSR_und
------	------------------	------------------	------------------	------------------	------------------

▤ = 分组寄存器

图 2-7 ARM 的寄存器组织

##### ① R0~R15

ARM 中的通用寄存器是 R0~R15 (R15 也是 PC)。它们可以被分为以下 3 类:

- 未分组的寄存器 R0~R7。对于所有的模式, R0~R7 所对应的物理寄存器都是相同的。这 8 个寄存器是真正意义上的通用寄存器, ARM 体系结构中对它们没有作任何特殊的假设, 它们的功能都是等同的。在中断或者异常处理程序中一般都需要对这几个寄存器进行保存。
- 分组的寄存器 R8~R14。程序访问的物理寄存器取决于当前的处理器模式。若要

访问特定的物理寄存器而不依赖于当前的处理器模式,则要使用规定的名字。R8~R12 有两组物理寄存器:一组是 FIQ 模式;另一组是除 FIQ 以外的其他模式。R13~R14 有 6 个组的物理寄存器,一组用于用户模式和系统模式,其他 5 组分别用于 5 种异常模式。R13 (也被称为 SP 指针) 被用做栈指针,通常在系统初始化时需要对所有模式的 SP 指针赋值;当 CPU 在不同的模式时栈指针会被自动切换成相应模式下的值。R14 有两个用途:一是在调用子程序时用于保存调用返回地址;二是在发生异常时用于保存异常返回地址。

- 程序计数 R15 (或者 PC): 寄存器 R15 用作程序计数器 (PC)。R15 虽然可以用作通用寄存器,但是有一些指令在使用 R15 时有一些特殊限制,若不注意,执行的结果将是不可预料的。

## ② CPSR

CPSR (当前程序状态寄存器) 在所有的模式下都是可以读/写的。它主要包含条件标志、中断标志、当前处理器的模式、其他的一些状态和控制标志。

31	30	29	28	27		8	7	6	5	4	3	2	1	0
N	Z	C	V	DNM(RAZ)			I	F	T	M4	M3	M2	M1	M0

CPSR 的格式如下:

- 条件标志包括 N, Z, C, V。
  - ◆ N——Negative, 负标志。
  - ◆ Z——Zero, 零标志。
  - ◆ C——Carry, 进位标志。
  - ◆ V——Overflow, 溢出标志。
- 中断标志包括 I, F。
  - ◆ I——置 1 表示禁止 IRQ 中断的响应,置 0 表示允许 CPU 响应 IRQ 中断。
  - ◆ F——置 1 表示禁止 FIQ 中断的响应,置 0 表示允许 CPU 响应 FIQ 中断。
- ARM/Thumb 控制标志: T。
  - ◆ 置 0 表示执行 32 bits 的 ARM 指令。
  - ◆ 置 1 表示执行 16 bits 的 Thumb 指令。
- 模式控制位 M0~M4, 见表 2-4。

## (5) 指令集

一个 CPU 的指令集是硬件和软件之间的一个重要的分水岭。根据分层的思想,指令集向上要支持编译器,向下要方便硬件的设计实现。ARM 是典型的 RISC 体系,根据 RISC



的设计思想，其指令集的设计应该尽可能地简单。和 CISC 体系相比，它可以通过一系列简单的指令来实现复杂指令的功能。

表 2-4 模式控制位及可用寄存器

M[4:0]	模 式	可用寄存器
0b10000	User	PC, R14~R0, CPSR
0b10001	FIQ	PC, R14_fiq~R8fiq, R7~R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12~R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12~R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12~R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12~R0, CPSR, SPSR_und
0b11111	System	PC, R14~R0, CPSR(ARM architecture v4 and above)

ARM 的指令集包括 6 种典型的指令：分支指令，如 B, BL 等；数据处理指令，如 ADD, SUB, AND 等；转移指令，如 MRS, MSR 等；Load-Store 数据移动指令，如 LDR 等；协处理器指令，如 LDC, STC 等；异常处理指令，如 SWI 等。

ARM 指令集是一个非常优秀的指令集。它有以下特点：

- 所有 ARM 指令都是 32 位定长，在内存中以 4 字节边界保存（地址最后两位为 0），这样方便译码电路和流水线的实现。ARM 内核一般也支持一种 16 位的指令集 Thumb。Thumb 指令集的功能是 32 位 ARM 指令集的功能子集，它在处理器中仍然要扩展为标准的 32 位 ARM 指令来运行。用户采用 16 位 Thumb 指令集最大的好处就是可以获得更高的代码密度和降低功耗。
- Load-Store 体系结构。ARM 指令集属于 RISC 体系。RISC 体系的特征就是：一般指令只能把内部寄存器和立即数作为操作数，只有 Load-Store 类型的数据移动指令才可以访问内存，在内存和寄存器之间转移数据。
- 由于硬件上有桶形（barrel）移位器，所以 ARM 可以在一条指令中用一个指令周期完成一个移位操作和一个 ALU（算术逻辑）操作。
- 所有指令都可以条件执行，这是由其指令格式决定的，如下所示：

31	cond	28	27	0
----	------	----	----	---

任何指令的高 4 位都是条件指示位，根据 CPSR 中的 N, Z, C, V 决定该指令是否执行。这样可以方便高级语言的编译器设计，很容易实现分支和循环。

- 有功能很强的一次加载和存储（Load-Store）多个寄存器的指令：LDM 和 STM。这样，当发生过程调用或中断处理时，只用一条指令就能把当前多个寄存器的内

容保护到内存堆栈中。

### (6) 异常

异常是由内部或外部原因引起。当异常发生时, CPU 自动到指定的向量地址读取指令或地址并且执行。

对 X86 CPU, 当有异常发生时, CPU 是到指定的向量地址读取要执行的程序的地址, 跳转到相应的地址并执行; 而对于 ARM CPU, 当有异常发生时 CPU 是到向量地址的地方读取指令并执行, 也就是 ARM 的向量地址处存放的是一条指令 (一般是一条跳转指令)。

ARM 将引起异常的类型分为 7 种, 如表 2-5 所列。

表 2-5 ARM 的异常类型

异常类型	模 式	优先级	一般向量地址	高向量地址
Reset	Supervisor	1	0x00000000	0xFFFF0000
Undefined Instruction	Undefined	6	0x00000004	0xFFFF0004
Software Interrupt	Supervisor	6	0x00000008	0xFFFF0008
Prefetch Abort	Abort	5	0x0000000C	0xFFFF000C
Data Abort	Abort	2	0x00000010	0xFFFF0010
IRQ(interrupt)	IRQ	4	0x00000018	0xFFFF0018
FIQ(fast interrupt)	FIQ	3	0x0000001C	0xFFFF001C

### (7) 内存和 I/O 地址

ARM 的寻址空间是线性地址空间, 最大  $2^{32}\text{B}=4\text{GB}$ 。ARM 支持大端和小端的内存数据方式, 可以通过硬件的方式设置端模式。

I/O 端口的编址方法即地址安排方式有两种: I/O 映射编址和存储器映射编址。

#### ① I/O 映射编址

如图 2-8 所示, I/O 映射编址采用 I/O 端口与内存单元分开编址, 互不影响。I/O 单元与内存单元都有自己独立的地址空间。通过专门的输入指令 (IN) 和输出指令 (OUT) 来完成 I/O 操作。

其优点是 I/O 单元不占用内存空间, 易区分 I/O 程序; 缺点只用 I/O 指令访问 I/O 端口, 功能有限且要采用专用 I/O 周期和专用 I/O 控制线, 使微处理器复杂化。X86 体系的微处理器大多采用 I/O 映射编址方式。

#### ② 存储器映射编址

如图 2-9 所示, 存储器映射编址采用 I/O 端口的地址与内存地址统一编址方式, I/O 单元与内存单元在共享同一地址空间。这种编址方式不区分存储器地址空间和 I/O 端口地址空间, 把所有的 I/O 端口都当作是存储器的一个单元对待, 每个接口芯片都安排一个或几

个与存储器统一编号的地址号。也不设专门的输入/输出指令，所有传送和访问存储器的指令都用来对 I/O 端口操作。

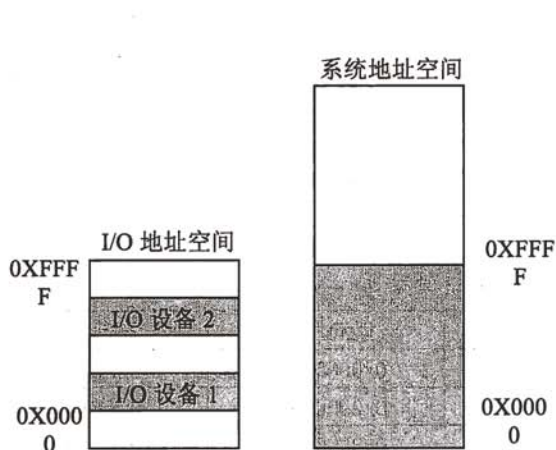


图 2-8 I/O 映射编址方式

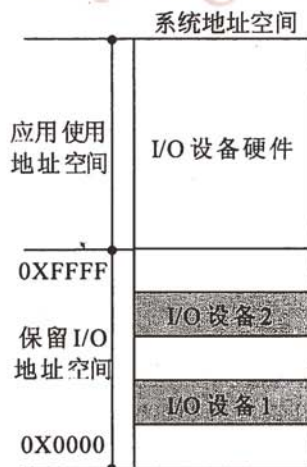


图 2-9 存储器映射编址方式

其优点是可采用丰富的内存操作指令访问 I/O 单元，无需单独的 I/O 地址译码电路，无需专用的 I/O 指令；缺点是外设占用内存空间，不易区分 I/O 程序，如 ARM 微处理器大多采用存储器映射编址方式。

#### (8) Intel 公司的 XScale PXA270 处理器

Intel XScale 微结构体系提供了一种全新的、高性价比、低功耗且基于 ARMv5TE 体系结构的解决方案，支持 16 位 Thumb 指令和 DSP 扩充，是 Intel 公司的 StrongARM 系列处理器的升级换代产品。Intel XScale CPU 内核采用带有一个增强型存储器管道的超级流水线 RISC 处理器体系结构，这种微体系结构在 ARM 核的周围提供了指令与数据存储器管理单元，指令、数据和微小数据 cache，写缓冲、全缓冲、挂起缓冲和分支目标缓冲器，电源管理，性能监控、调试和 JTAG 单元以及协处理器接口，MAC 协处理器和内核存储总线。其特点如下：

- Intel 7~8 级超流水线结构带来的高性能和超低功耗。
- Intel 动态电压管理，可以动态管理芯片电压和时钟频率，让使用者可以在功耗和性能上取得平衡。
- Intel 媒体处理技术，可有效处理多媒体指令。
- 128 个跳转指令目的地址缓存可存储跳转指令的目的地址，让指令预取和指令流水线获得更高效率。

- 32KB 数据缓存和指令缓存;
- 调试单元拥有硬件中断功能, 可存储 256 个断点位置。
- 64 位内核内存数据宽度, 可以让内核在 600MHz 时钟频率下获得 4.8GB/s 的高速数据流。

Intel PXA270 处理器是针对高端便携式手持设备及工业设备推出的一款高性能、低功耗、功能强大的嵌入式 SOC 微处理器产品, 它采用 Intel Xscale 微结构体系结构, 最高主频达 624MHz; 具有大小为 23×23mm、引脚间距为 1.0mm 的 PBGA 封装和大小为 13×13mm、引脚间距为 0.5mm 的 VF-BGA 封装的两种封装形式; 具备 Intel 的无线多媒体扩展技术 (Wireless MMX), 能够流畅的运行三维游戏和播放高质量的多媒体及视频文件; PXA270 的 Quick Capture 技术使其能够拍摄高达 400 万像素的图像和视频, 并支持低功耗、实时的回放处理; 支持 24 位色的 LCD 显示, 具有 256KB 的片上 SRAM 帧缓冲, 这和 Quick Capture 一起加速了图像的回放; 支持 Intel 专用的无线 SpeedStep 动态电源管理技术, 使处理器根据系统运行的不同电源状况, 自动切换工作频率和电压, 从而实现嵌入的、智能的电源管理; 此外还具有丰富的外围接口, 如 USB Host/Client、USB OTG、4 位的 SD I/O、MMC/SD 卡、CMOS/CCD、OTG、IDE、LAN、Memory Stick、USIM 卡接口、Keypad 控制器等。PXA270 可以广泛应用于 PDA、智能手机、PMP 等嵌入式产品中。其结构框图如图 2-10 所示。

PXA270 处理器加入了 wireless MMX 技术和 SpeedStep 动态电源管理技术, 不但增强了 PXA270 的媒体处理能力, 而且极大的降低了系统功耗, 延长便携产品的电池寿命。PXA270 的 Quick Capture 技术最大可支持 400 万像素的 CCD 摄像头, 数码摄像功能强大; 且具备 3D 加速功能, 满足了游戏应用; 支持 LAN 接口, 可以扩展网络应用。PXA270 处理器作为一款性能强劲的嵌入式处理器, 被广泛应用于手机、高端 PDA 等高端便携式手持设备及工业设备中。

下面将详细分析 PXA270 处理器的一些特性:

- Wireless MMX

Pentium's MMX 是一种基于 Intel MMXT 先进的多媒体指令集 MMX 技术, 使基于 PXA270 的嵌入式设备在拥有多媒体处理能力的同时, 能够最大限度降低系统功耗; 另一方面, 也有助于软件开发商提供游戏、MPEG4 视频文件及语音识别等应用服务。该款芯片把 X86 体系结构 Pentium 4 系列上的多媒体扩展功能引入了 Xscale 芯片组的产品线中, 用户通过该无线多媒体扩展技术(MMX)可以在嵌入式设备上播放高质量的视频, 流畅地运行三维游戏。

PXA270 处理器也支持无线 MMX 技术。无线 MMX 技术是一种加快处理器多媒体处理的新指令, 由于无线 MMX 类似于桌面处理器的 MMX 技术, 因此, 针对桌面应用且为

MMX 优化的程序可以很好的运行在 PXA27x 处理器上, 并且最终的 MMX 优化方式同桌面 MMX 的优化技术几乎是一样的, 这便于程序开发商将很多 PC 上运行的程序移植到基于 PXA270 处理器的系统上来。

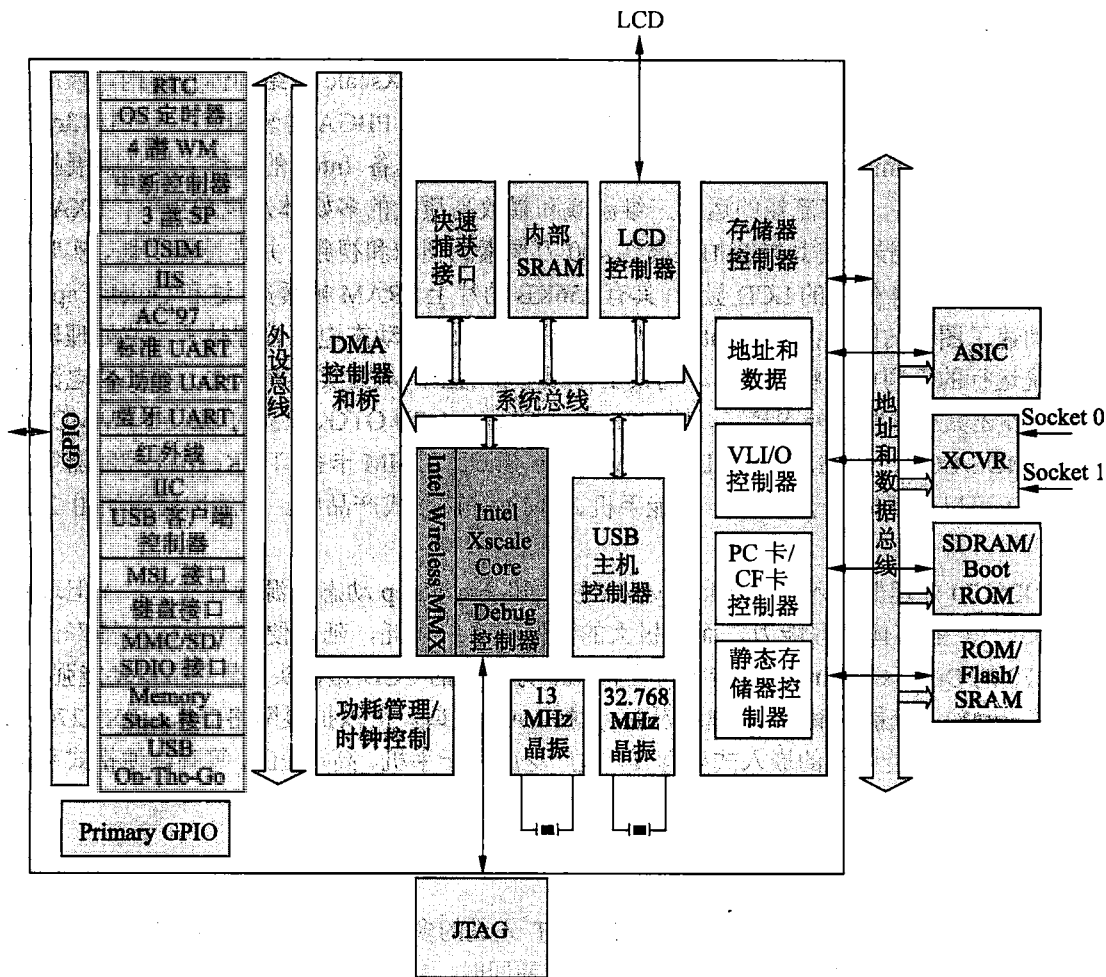


图 2-10 PXA27x 芯片内部结构

#### • Quick Capture

Quick Capture 作为成像设备与无线设备间的接口, 有助于改进图像质量, 降低产品整体成本。该项技术包括快速浏览、快速拍照和快速视频拍摄 3 种操作模式。该技术使 Intel PXA27x 系列处理器可以支持 400 万像素数码镜头, 并能提供最大 416Mbit 的数据传输

速率。

PXA270 处理器的 Quick Capture 技术最高可以支持 2048×2048 像素分辨率的 400 万像素的图像拍摄和处理,同时具有每秒最高 25MB/s 的传输和处理速度,使用户能够进行高速实时回放和传输等操作。图 2-11 为 Quick Capture 的处理流程图。

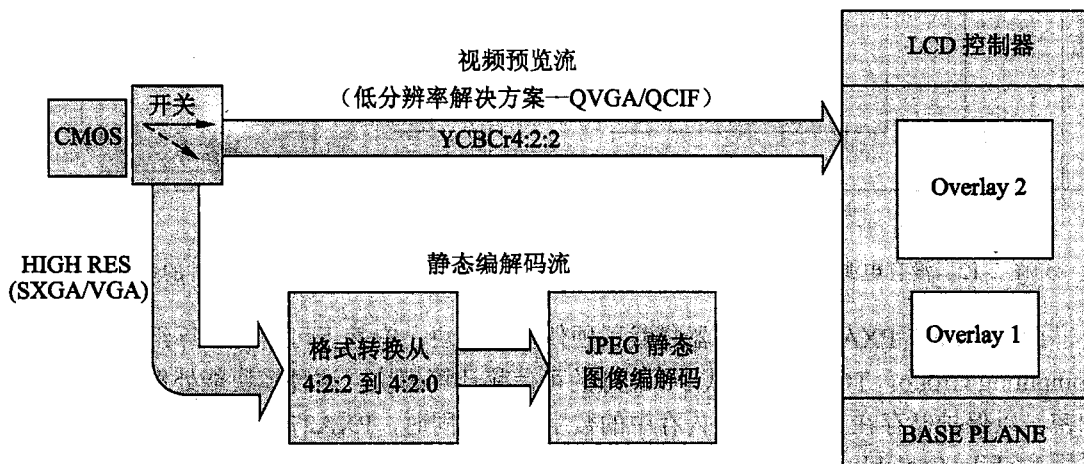


图 2-11 Quick Capture 的处理流程

此外, PXA270 处理器还提供了对扩展 LCD 的支持,这种技术允许处理器支持第二个 24 位真彩色的 LCD 屏幕,同时其包括的 256KB SRAM 帧缓冲使两个屏幕都可以高速正常地显示图像。

#### • SpeedStep

SpeedStep 技术原用于 Intel 移动处理器,这种技术用通俗的语言表述就是系统需要多高的主频,它就调节到多高的频率,系统不需要时,它就将处理器主频调节到最低,从而降低系统功耗。SpeedStep 技术可以智能地切换空闲、待机和深层睡眠 3 种低功耗状态,以提高动态电压管理性能,在保证 CPU 性能的情况下,最大限度地降低嵌入式设备的功耗。

PXA270 处理器支持专用的无线 SpeedStep 技术,这种技术可以使处理器根据系统运行的不同电源状况,自动切换工作频率和电压。虽然之前的 PXA 系列处理器在运行过程中也能够改变处理器的时钟频率,但采用无线 SpeedStep 技术的 PXA270 处理器,却能够结合处理器的工作频率,在 26MHz~624MHz (最高) 之间自由调节,改变电压,实现低电能消耗。也就是说,在系统完全空闲时, PXA27x 可以运行在 26MHz 的主频下,此时它的功耗将低于 0.1 mA! 根据 Intel 公司的资料称,在启用无线 MMX 和无线 SpeedStep 技术之后可以节省 30%~77% 的功耗。图 2-12 和图 2-13 为是否具有电源管理软件的情况下的耗能

对比。

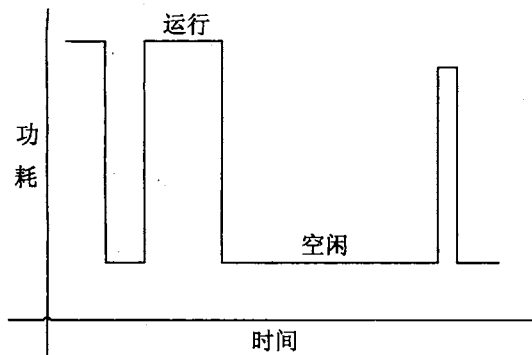


图 2-12 没有电源管理软件的系统耗能

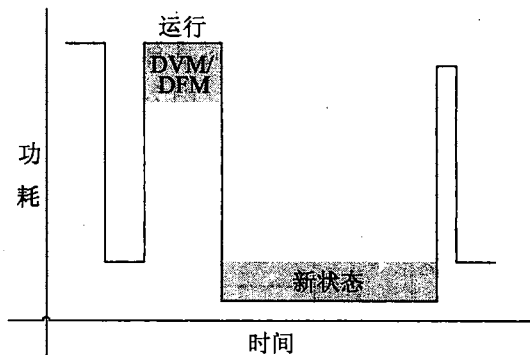


图 2-13 具有电源管理软件的系统耗能

此外，Intel PXA 270 处理器搭载了硬件安全模块，它基于可信赖计算组织（Trusted Computing Group, TCG）为台式 PC 制订的可信赖计算模块。这种模块能处理硬件速度的加密，以保护储存在 32KB 片上内存中的数字密钥。同时，PXA270 也是 Intel 公司第一款采用 Mobile Scalable Link (MSL) 接口的应用处理器。MSL 是一种基带接口，能在 1、2 或 4 位带宽上支持 416 Mb/s 的数据速率。具有 1.8V 多路和带缓存 I/O 通道。

## 2. MIPS 系列

MIPS 是世界上很流行的一种 RISC 处理器。MIPS (Microprocessor without Interlocked Piped Stages) 的意思是“无互锁流水级的微处理器”，其机制是尽量利用软件办法避免流水线中的数据相关问题。它最早是在 20 世纪 80 年代初期由美国斯坦福大学的 Hennessy 教授领导的研究小组研制出来的。MIPS 公司的 R 系列处理器就是在此基础上开发的 RISC 工业产品的微处理器。这些系列产品已被很多在计算机公司采用来构成各种工作站和计算机系统。

从 MIPS 处理器发明到现在的这 20 多年里，MIPS 处理器以其高性能的处理能力被广泛应用于宽带接入、路由器、调制解调设备、电视、游戏、打印机、办公用品、DVD 播放等广泛的领域。

和 ARM 公司一样，MIPS 公司本身并不从事芯片的生产活动（只进行设计），不过其他公司如果要生产该芯片，则必须得到 MIPS 公司的许可。

MIPS 32 位处理器内核有如下几种：

- M4K™ 系列针对多 CPU 集成的 SOC，应用领域为下一代消费类产品、下一代网络和宽带产品。
- M4K™ 系列包含 4Kp™、4Kc™ 内核，针对 SOC 系统优化，其内存、指令缓存和

数据缓存都可以根据具体应用调整大小。

- M4K™ 系列包含 4KEp™、4KEm™ 和 4KEc™ 内核；和 4K™ 系列类似，但能提供更高性能，在同样时钟频率下指令执行周期更短。
- 4KS™ 系列包含 4KSc™ 和 4KSd™ 内核，是针对数据通信的应用。其特点是采用了 SmartMIPS™ 结构，拥有反黑客的特性，可以让数据加密更加快速，在网络处理、智能卡、机顶盒等方面有广泛应用。
- Pro Series™ 系列包含 M4K Pro™、4KE Pro™、4KEm Pro™、4KEc Pro™ 和 4KSd Pro™ 内核，该系列内核包含了空前的特性：允许 SOC 的设计者创造自己的 CorExtend™ 扩展指令集。这样可以根据具体应用设计出性能更好，效率更高的产品。
- 24K™ 系列针对图形、JAVA 应用，包含了最快的浮点乘法器，也支持 CorExtend™ 扩展指令集，是数字电视、机顶盒和 DVD 等多媒体应用的理想选择。

下面将重点介绍 MIPS 32 和 MIPS 64 的体系结构。

### 1) MIPS 体系的组成

#### (1) MIPS 指令集体系 ISA(MIPS Instruction Set Architecture)

从最早的 MIPS I ISA 开始发展，到 MIPS V ISA，再到现在的 MIPS 32 和 MIPS 64 结构，其所有版本都是与前一个版本兼容的。在 MIPS III 的 ISA 中，增加了 64 位整数和 64 位地址；在 MIPS IV 和 MIPS V 的 ISA 中，增加了浮点数的操作等。

MIPS 32 和 MIPS 64 体系是为满足高性能、成本敏感的需求而设计的。MIPS 32 体系是基于 MIPS II ISA 的，并从 MIPS III, MIPS IV 和 MIPS V 中选择一些指令以增强数据及代码的有效操作。MIPS 64 体系是基于 MIPS V ISA 并与 MIPS 32 体系兼容的。MIPS 32 和 MIPS 64 体系都将特权环境 (privileged environment) 引入体系，以满足操作系统或其他核心软件的需要。同时 MIPS 32 和 MIPS 64 体系支持 ASE (MIPS Application Specific Extensions)、UDI (User Defined Instructions) 等协处理器，以满足特定领域的应用需求。

#### (2) MIPS 特权资源体系 PRA

MIPS 32 和 MIPS 64 的 PRA (MIPS Privileged Resource Architecture) 定义了一组指令，其中大多数指令只能在特权模式下使用。有些指令在非特权模式下也是可见的，如虚拟内存布局。PRA 提供了管理处理器资源所必需的机制，如虚存、cache、异常和用户的上下文等。

#### (3) MIPS 特定应用扩展 ASE

MIPS 32 和 MIPS 64 体系支持可选的特定应用的扩展 ASE (MIPS Application Specific Extensions)。ASE 是对基本体系的扩展，不承担体系中指令的实现，是在 ISA 和 PRA 基础上完成特定领域应用的需要。

#### (4) MIPS 用户定义指令集 UDI

除了支持 ASE 外，MIPS 32 和 MIPS 64 体系还提供专门的指令，即用户定义指令集



UDI (MIPS User Defined Instructions)。这些指令的功能是在具体实现时进行定义的。

## 2) 数据类型

MIPS 处理器定义了下列数据格式:

- 位 bit (b)。
- 字节 Byte (8 bit, B)。
- 半字 Halfword (16 bits, H)。
- 字 Word (32 bits, W)。
- 双字 Doubleword (64 bits, D)。

浮点处理器定义了下列数据格式:

- 32 位的单精度浮点数 (.fmt type S)。
- 32 位的单精度浮点单对数 (.fmt type PS)。
- 64 位双精度浮点数 (.fmt type D)。
- 32 位字固定数 (.fmt type W)。
- 64 位长固定数 (.fmt type L)。

## 3) 协处理器

MIPS 体系定义了 4 个协处理器 CP0, CP1, CP2, CP3。

- CP0 是在 CPU 芯片内的, 支持虚存、管理异常和处理核心态与用户态的切换, 控制 cache 系统, 提供诊断控制和错误恢复机制, 通常被称为系统控制协处理器 (System Control Coprocessor, SSC)。
- CP1 保留给 FPU。
- CP2 可用于专门的实现。
- CP3 保留给 MIPS 64 版本 1 和所有体系版本 2 的 FPU。

## 4) MIPS 32 体系定义了下列 CPU 寄存器。

- 32 个 32 位通用寄存器 GPRs (General Purpose Registers): r0~r31。
- 一对专门的寄存器 HI 和 LO, 用于存放整数乘、除和乘加运算的结果。
- 程序计数器 PC。

## 3: PowerPC

### 1) PowerPC 的特点

PowerPC 是 Motorola (现为 Freescale) 公司的产品。PowerPC RISC 处理器实现性能增强的最主要原因就在于修改了指令处理设计, 它比传统处理器的指令处理效率要高得多。它完成一个操作所需的指令数比 CISC 处理器要多, 但完成操作的总时间却减少了。这主要是因为前者采用了超标量处理器设计和调整内存缓冲器。PowerPC 内核的主要特点如下:

- 独特分支处理单元可以让指令预取效率大大提高, 即使指令流水线上出现跳转指

令,也不会影响到其运算单元的运算效率。

- 超标量 (Superscale) 设计。分支单元、浮点运算单元和定点运算单元,每个单元都有自己独立的指令集并可独立运行。
- 可处理“字节非对齐”的数据存储。
- 同时支持大端小端 (Big/Little-Indian) 数据类型。

## 2) PowerPC 的技术特点

### (1) 分支处理器

RISC 处理器设计的一个主要目标就是让处理器在一个时钟周期内至少处理一条指令。然而系统的内存往往达不到这个速度,这就使处理器不能及时从内存中读出需要执行的指令。所以在 RISC 系统中,处理器一般会同时读入多条指令到它们的缓存或是指令流水线上,这种技术就叫指令预取。当处理器执行到跳转指令时,由于下一条指令可能在系统内存的任何一个地址,这样指令预取就会失败,处理器不得不到内存中读取相关指令,而处理器的其他部分就只能等新的指令被读取,这样就降低了处理器效率。

PowerPC RISC 处理器设计了多级内存高速缓冲区,以便让那些正在访问(或可能会被访问)的数据和指令总是存储在调整内存中。这种内存分层和内存管理设计令系统的内存访问性能非常接近调整内存,但其成本却与低速内存相近。而且 PowerPC 还引入了独立的分支处理器来进一步解决这个问题,这个处理单元在读入指令队列后,会找出其中的跳转指令,然后预取跳转指令所指向的新的内存地址的指令,这样就大大提高了指令预取的效率。

### (2) 超标量设计

PowerPC 的设计者并不满足处理器只是在一个时钟周期处理一条指令,他们希望处理器能同时处理更多的指令。在 PowerPC 内部,集成了多个处理器,这些处理器可以并行独立工作,这样就可以在一个时钟周期执行多条指令。一个标准的 601 处理器中便集成了一个定点处理器、一个浮点处理器和一个分支处理器,这种超标量设计提供了允许多条指令同时运行的多处理流水线。显然,这种指令的重叠程度取决于指令的顺序和种类。

### (3) 字节非对齐操作的兼容

大多数 RISC 处理器要求对存储器件的访问必须是字节对齐的,也就是说对任何 16 位的数据操作必须是在偶数地址上进行,而对 32 位的数据操作,地址必须是 4 的倍数。而 PowerPC 却可以处理字节非对齐的存储器访问,这种特性可以让它兼容许多从 CISC 处理器移植过来的指令和数据结构。不过需要指出的是,字节非对齐的操作会降低处理器的性能,所以如果能让指令执行得更快,最好还是把数据类型优化成字节对齐的。

### (4) 字节顺序的兼容

PowerPC 和 68K 系列处理器都采用大端模式的字节顺序,因此 PowerPC 可以很方便

地兼容 68K 系列处理器和数据结构。但为了能和其他小端模式的处理器或是外设进行数据交换, PowerPC 也可以工作在小端模式下, 它可以通过一些特殊指令访问小端模式的数据, 不过在这种情况下, PowerPC 就不能访问非字节对齐的数据了。

### 2.1.5 DSP 处理器的结构和特点

在 PC 上所使用的处理器, 如早期的 Intel 8086 处理器, 大约 90% 的运算量都在做加法运算, 每一个加法运算需 3 个时钟, 而一个乘法运算即需要 134~160 个时钟, DSP 处理器的就是为了满足庞大乘法运算的需要而产生的。

数字信号处理器与一般常见微处理器的不同, 主要是在于总线与内存空间上体系结构的差异。一般用途的微处理器大多采用冯·诺伊曼体系结构 (von Neumann Architecture), 这种体系结构的程序与指令存储于相同的内存空间, 处理器核心利用相同的总线存取内存中的指令与数据, 然后进行数据处理, 指令与数据的存取无法同时被处理, 也就是当处理器要执行指令时, 必须先将指令由内存中读取进来译码, 接着再由内存中读取运算数据, 而后进行数据的运算, 如此必须花费许多数据传输时间。

数字信号处理器拥有哈佛体系结构 (Harvard Architecture), 这种结构拥有不同的指令与运算数据总线, 使数字信号处理器可以在同一时间由内存中读取指令与运算数据, 加速进行运算, 以减少运算时所需浪费的时间。一般的 DSP 处理器中含有 3 条总线: 一条指令总线和两条运算数据总线。这使数字信号处理器一次可以读取一条指令和两个运算数据, 从而大大提高了数据处理的能力, 但同时设计处理器时的复杂性也相应增加。

为了加快数字信号处理器的运算速度, 数字信号处理器设计了许多特殊的体系结构, 并且强化了它的并行化处理能力。例如, 一般的数字信号处理器都有一个乘法器与一个加法器, 数字信号处理器允许在同一个时间内处理一个乘法运算及一个加法运算; 有的数字信号处理器能够使用蝴蝶方式 (butterfly) 并行地进行傅立叶转换, 其中包含了移位器, 能够在不使用加法器的情况下, 进行数据位的移位, 从而加速了处理的速度。而一般微处理器在进行类似运算时, 计算数据传输所需要依据的内存地址, 比处理数据运算所花的时间还多。例如在 Intel 8086 处理器中, 一个加法运算仅需 3 个时钟, 但是处理执行命令的地址运算即需要 5~12 个时钟。数字信号处理器中设计了矩阵数据的计算硬件体系结构, 包含了特别的算术单元来作为地址产生的用途, 所以在数字信号处理器中, 地址的计算并不需要额外的加法运算时间。

在数字信号处理器上最基本的运算功能就是乘法与加法运算。除此之外, 数字信号处理器还用于一些常见算法的实现, 如有限脉冲响应滤波器 (Finite Impulse Responsefilter, FIR)、无限脉冲响应滤波器 (Infinite Impulse Responsefilter, IIR)、离散傅立叶 (Discrete Fourier Transforms) 及离散余弦转换 (Discrete Cosine Transforms) 等, 数字信号处理器大

多已经内置硬件体系结构，可以在短时间之内处理这些的加法与乘法问题。

### 1. 典型的数字信号处理器介绍

下面以 TI 公司的 TMS320LF2407A 为例，对其内部结构和组成进行介绍。其结构框图如图 2-14 所示。

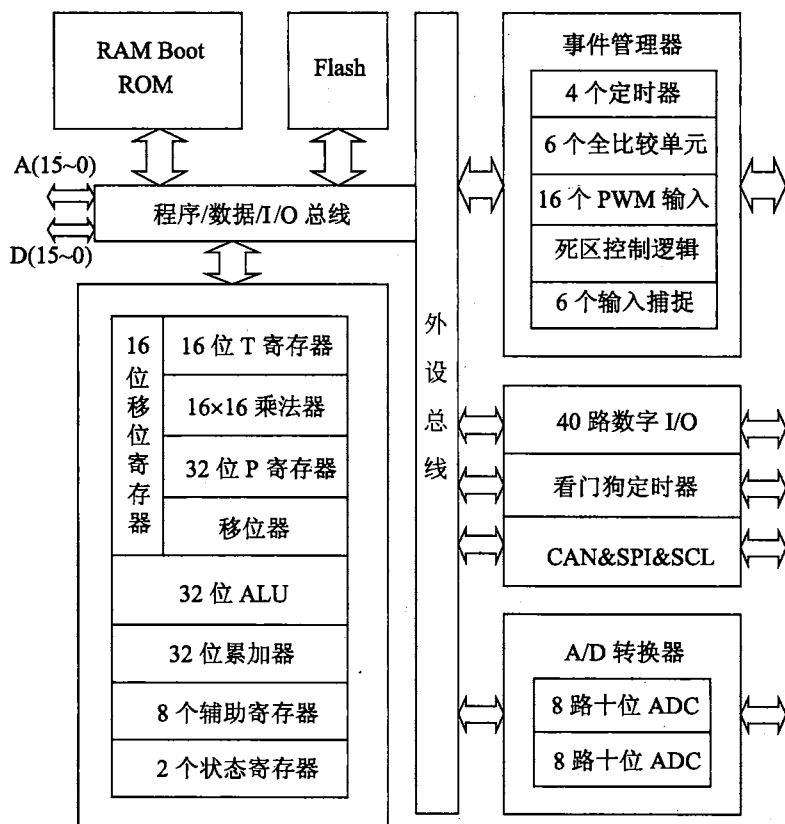


图 2-14 TMS320LF2407A 的模块框图

TMS320LF2407A 是 TI 公司的一款应用于控制领域的定点 DSP 芯片，它采用了高性能静态 CMOS 技术，使供电电压降为 3.3V，减小了控制器的功耗；运算速度快，40MIPS 的执行速度使指令周期缩短到 25ns（40MHz），从而提高了控制器的实时控制能力；集成了 32KB 的 Flash、2.5KB 的 RAM；外围接口丰富，功能强大，500ns 转换时间的 A/D 转换器，片上事件管理器提供了 PWM 接口和 I/O 功能，此外还提供看门狗电路、SPI、SCI 和 CAN 控制器等；价格便宜，应用也越来越广泛。

### 1) CPU 及总线结构

TMS320LF2407A 的 CPU 是基于 TMS320C2XX 的 16 位定点低功耗内核。体系结构采用四级流水线技术加快程序的执行,可在一个处理周期内完成乘法、加法和移位运算。其中央算术逻辑单元 (CALU) 是一个独立的算术单元,它包括一个 32 位算术逻辑单元 (ALU)、一个 32 位累加器、一个  $16 \times 16$  位乘法器 (MUL) 和一个 16 位桶形移位器,同时乘法器和累加器内部各包含一个输出移位器。完全独立于 CALU 的辅助寄存器单元 (ARAU) 包含 8 个 16 位辅助寄存器,其主要功能是在 CALU 操作的同时执行 8 个辅助寄存器 (AR7 至 AR0) 上的算术运算。两个状态寄存器 ST0 和 ST1 用于实现 CPU 各种状态的保存。

TMS320LF2407A 采用增强的哈佛结构,芯片内部具有 6 条 16 位总线,即程序地址总线 (PAB)、数据读地址总线 (DRAB)、数据写地址总线 (DWAB)、程序读总线 (PRDB)、数据读总线 (DRDB)、数据写总线 (DWEB),其程序存储器总线 and 数据存储器总线相互独立,支持并行的程序和操作数寻址,因此 CPU 的读/写可在同一周期内进行,这种高速运算能力使自适应控制、卡尔曼滤波、神经网络、遗传算法等复杂控制算法得以实现。

### 2) 存储器配置

TMS320LF2407A 地址映像被组织为 3 个可独立选择的空间:程序存储器 (64KB)、数据存储器 (64K)、输入/输出 (I/O) 空间 (64KB)。这些空间提供了共 192KB 的地址范围。

其片内存储器资源包括:  $544\text{B} \times 16$  位的双端口数据/程序 DARAM、 $2\text{KB} \times 16$  位的单端口数据/程序 SARAM、片内  $32\text{KB} \times 16$  位的 Flash 程序存储器、 $256\text{B} \times 16$  位片上 Boot ROM、片上 Flash/ROM 具有可编程加密特性。

### 3) 事件管理器模块

TMS320LF2407A 包含两个专用于电机控制的事件管理器模块 EVA 和 EVB,每个事件管理器模块包括通用定时器 (GP)、全比较单元、正交编码脉冲电路以及捕获单元。

#### (1) 通用定时器

TMS320LF2407A 共有 4 个 16 位通用定时器,可用于产生采样周期,作为全比较单元产生 PWM 输出以及软件定时的时基。通用定时器有 4 种可选择的操作模式:停止/保持模式、连续增计数模式、定向增/减计数模式和连续增/减计数模式。每个通用定时器都有一个相关的比较寄存器 TxCMPR 和一个 PWM 输出引脚 TxPWM。每个通用定时器都可以独立地提供一个 PWM 输出通道,产生对称的或非对称的 PWM 波形。因此,4 个通用定时器最多可提供 4 路 PWM 输出。

#### (2) 全比较单元

每个事件管理器模块有 3 个全比较单元 (1、2 和 3 (EVA); 4、5 和 6 (EVB)), 每个

比较单元各有一个 16 位比较寄存器 CMPRx, 各有两个 CMP/PWM 输出引脚, 可产生两路 PWM 输出信号控制功率器件, 其输出引脚极性由控制寄存器 (ACTR) 的控制位来决定, 根据需要, 选择高电平或低电平作为开通信号, 通过设置 T1 为不同工作方式, 可选择输出对称 PWM 波形、非对称 PWM 波形或空间矢量 PWM 波形。

死区控制单元 (DBTCON) 用来产生可编程的软件死区, 每个全比较单元的两路 CMP/PWM 输出控制的功率器件, 在相隔两次开启周期间没有重叠, 最大可编程的软件死区时间达 16 $\mu$ s。

### (3) 正交编码脉冲电路

正交编码脉冲 (QEP) 电路可以对引脚 CAP1/QEP1 和 CAP2/QEP2 上的正交编码脉冲进行解码和计数, 可以直接处理光电编码盘的双路正交编码脉冲。正交编码脉冲包含两个脉冲序列, 有变化的频率和四分之一周期 (90°) 的固定相位偏移, 对输入的两路正交信号进行鉴相和 4 倍频。通过检测 2 路信号的相位关系可以判断电机的正/反转, 并据此对信号进行加/减计数, 从而得到当前的计数值和计数方向, 即电机的角位移和转向, 电机的角速度可以通过脉冲的频率测出。

### (4) 捕获单元

捕获单元用于捕获输入引脚上信号的跳变, 两个事件管理器模块总共有 6 个捕获单元。EVA 模块有 3 个捕获单元引脚 CAP1、CAP2 和 CAP3, 它们可以选择通用定时器 1 或 2 作为时基, 但 CAP1 和 CAP2 一定要选择相同的定时器作为时基; EVB 模块也有 3 个捕获单元引脚 CAP4、CAP5 和 CAP6, 它们可以选择通用定时器 3 或 4 作为时基, 但 CAP4 和 CAP5 一定要选择相同的定时器作为时基。每个单元各有一个两级的 FIFO 缓冲堆栈。当捕获发生时, 相应的中断标志被置位, 并向 CPU 发中断请求。

### 4) 片内集成外设

TMS320LF2407A 片内集成了丰富的外设, 大大减少了系统设计的元器件数量。

#### (1) 串行通信接口

TMS320LF2407A 设有一个异步串行外设通信接口 (SCI) 和一个同步串行外设通信接口 (SPI), 用于与上位机、外设及多处理器之间的通信。SCI 即通用异步收发器 (UART) 支持 RS-232 和 RS-485 的工业标准全双工通信模式, 用来与上位机的通信; SPI 可用于同步数据通信, 典型应用包括 TMS320LF2407A 之间构成多机系统和外部 I/O 扩展, 如显示驱动。

#### (2) A/D 转换模块

包括两个带采样/保持的各 8 路 10 位 A/D 转换器, 具有自动排序能力, 一次可执行最多 16 个通道的自动转换, 可工作在 8 个自动转换的双排序器工作方式或一组 16 个自动转换通道的单排序器工作方式。A/D 转换模块的启动可以有事件管理器模块中的事件源启动、



外部信号启动、软件立即启动等 3 种方式。

### (3) CAN 总线 (Controller Area Network, 控制器局域网)

CAN 是现场总线的一种, 主要用于各种设备的监测及控制。TMS320LF2407A 片上 CAN 控制器模块是一个 16 位的外设模块, 该模块完全支持 CAN2.0B 协议, 6 个邮箱(其中 0、1 用于接收; 4、5 用于发送; 2、3 可配置为接收或发送)每次可以传送 0~8 个字节的数据, 具有可编程的局部接收屏蔽、位传输速率、中断方案和总线唤醒事件、超强的错误诊断、自动错误重发和远程请求回应、支持自测试模式等功能。

CAN 总线通信可靠性高, 节点数有 110 个, 传输速度高达 1Mb/s (此时距离最长为 40m), 直接通信距离可达 10km (速率 5Kb/s 以下), 采用双绞线差分方式进行通信, 有很强的抗干扰能力。

### (4) 锁相环电路 (PLL) 和等待状态发生器

前者用于实现时钟选项; 后者可通过软件编程产生用于用户需要的等待周期, 以配合外围低速器件的使用。

### (5) 看门狗定时器与实时中断定时器

两者均为 8 位增量计数器, 前者用于监控系统软件和硬件工作, 在 CPU 出错时产生复位信号; 后者用于产生周期性的中断请求。

### (6) 外部存储器接口

可扩展为 192KB×16 位的最大可寻址存储器空间 (64KB 程序存储器、64KB 数据存储器、64KB I/O 空间)。

### (7) 数字 I/O

TMS320LF2407A 有 40 个通用、双向的数字 I/O 引脚, 其中大多数都是基本功能和一般 I/O 复用引脚。

### (8) JTAG 接口

由于 TMS320LF2407A 结构复杂、工作速度快、外部引脚多、封装面积小、引脚排列密集等原因, 传统的并行仿真方式已不适合于 TMS320LF2407A 的开发应用。TMS320LF2407A 具有符合 IEEE1149.1 规范的 5 线 JTAG (边界扫描逻辑) 串行仿真接口, 能够极其方便地提供硬件系统的在线仿真和测试。

### (9) 外部中断

有 5 个外部中断 (功率驱动保护、复位、不可屏蔽中断 NMI 及两个可屏蔽中断)。

### 5) 指令集

- 单指令循环和块循环操作。
- 块存储器搬移指令, 更便于程序和数据管理。
- 32 位长操作数指令。

- 同时读取 2 操作数和 3 操作数指令。
- 算术指令带并行存储和并行装入。
- 条件存储指令。
- 快速中断返回。

#### 6) 寻址方式

TMS320LF2407A 的指令集有 3 种基本的存储器寻址方式：立即寻址方式、直接寻址方式、间接寻址方式。

在立即寻址方式中，指令中包括了立即操作数。一条指令中可对两种立即数编码，一种是短立即数（3、5、8 或 9 位），另一种是 16 位的长立即数。短立即数指令编码为一个字长，16 位立即数的指令编码为两个字长。立即数寻址指令中在数字或符号常数前面加一个“#”号，表示立即数。

在直接寻址方式中，指令中包含数据存储器地址（dma）的低 7 位，这 7 位数据存储器地址作为地址偏移量，结合基地址（由数据页指针 DP 或堆栈指针 SP 给出）共同形成 16 位的数据存储器地址。使用这种寻址方式，用户可在不改变 DP 或 SP 的情况下，对一页内的 128 个存储单元随机寻址。采用这种寻址方式的好处是指令为单字指令，数据存储器地址的低 7 位放在指令字中。

间接寻址方式又可以分为单操作数寻址和双操作数寻址。双数据存储器操作数间接寻址类型为 \*ARx、\*ARx-、\*ARx+、\*ARx+0%，所用辅助寄存器只能是 AR2、AR3、AR4、AR5。其特点是：占用程序空间小，运行速度快，在一个机器周期内通过两个 16 位数据总线（C 和 D）读两个操作数。指令中 Xmem 表示从 DB 总线上读出的 16 位操作数，Ymem 表示从 CB 总线上读出的 16 位操作数。

#### 7) 主要应用

TMS320LF2407A 为高性能的控制提供先进、可靠、高效的信号处理与控制的平台，它将数字信号处理的运算能力与面向高性能控制的能力集于一体，可以实现用软件取代模拟器件，可方便地修改控制策略，修正控制参数，兼具故障监测、自诊断和上位机管理与通信等功能，将成为控制系统开发的主流处理器，可广泛应用于：工业电机驱动；能量交换器，如 UPS、通信电源；自动化系统，如电力控制、抗锁死制动；磁盘/光盘伺服控制和大容量存储产品；打印机、复印机和其他办公产品；仪器、仪表；机器人控制。

### 2. DSP 的发展方向

现在 DSP 产品的应用已扩大到人们的学习、工作和生活的各个方面，并逐渐成为电子产品更新换代的决定因素；而半导体技术和超大规模集成电路的发展为 DSP 的推陈出新提供了物理基础，用户的需求又为 DSP 的发展指出了方向。总体而言，DSP 技术将向以下几个方面发展。



### (1) 系统级集成

缩小 DSP 芯片尺寸始终是 DSP 的技术发展方向。当前的 DSP 多数基于精简指令集计算 (RISC) 结构, 这种结构的优点是尺寸小、功耗低和性能高。各 DSP 厂商纷纷采用新工艺, 改进 DSP 芯核, 并将几个 DSP 芯核、MPU 芯核、专用处理单元、外围电路单元和存储单元集成在一个芯片上, 成为 DSP 系统级集成电路。

### (2) 更高的运算速度

目前一般的 DSP 运算速度为 100MIPS, 即每秒钟可运算 1 亿条指令, 但由于电子设备的个人化和客户化趋势, DSP 必须追求更高更快的运算速度, 才能跟上电子设备的更新步伐。DSP 运算速度的提高主要依靠新工艺改进芯片结构。目前, TI 的 TM320C6X 芯片由于采用超长指令字 (全称为 Very Long Instruction Word, VLIW) 结构设计, 其处理速度已高达 2000MIPS。当前的 DSP 器件大都采用  $0.5\sim0.35\mu\text{m}$  的 CMOS 工艺。按照 CMOS 的发展趋势, DSP 的运算速度完全可能再提高 100 倍 (达到 1600GIPS)。

### (3) 可编程性

可编程 DSP 给生产厂商提供了很大的灵活性。生产厂商可在同一个 DSP 平台上开发出各种不同型号的系列产品, 以满足不同用户的需求。同时, 可编程 DSP 也为广大用户提供了易于升级的良好途径。人们已经发现, 许多微控制器能做的事情, 使用可编程 DSP 能做得更好更便宜, 例如冰箱、洗衣机这些原来装有微控制器的家电如今都已换成可编程 DSP 来进行大功率电机控制。

### (4) 支持高级编程语言的 DSP 开发软件

使用高级语言进行 DSP 软件开发可缩短软件开发的周期, 加快 DSP 产业的发展。

### (5) 并行处理结构

并行结构所带来的好处是显而易见的, 各 DSP 厂家纷纷在器件中引入并行机制, 主要分为片内并行和片间并行。

### (6) 功耗低

(略)

## 2.1.6 多核处理器的结构和特点

### 1. 多核处理器简介

双核或多核处理器早已有之, SOC、多媒体、网络等一些嵌入式处理器中都采用了多核技术, 但真正引人注目的是多核技术被引入到最高性能的通用处理器中, 因为无论是从技术的复杂度上说, 还是对未来处理器设计的影响来讲, 意义都极其重大。

在 2001 年, IBM 公司推出了基于双核的 Power 4 处理器, 这是世界上第一款采用多核技术的高性能服务器处理器; 随后 Sun 和 HP 公司都先后推出了基于双核体系结构的 UltraSPARC 及 PA-RISC 芯片。当前这些多核处理器主要应用于对提高性能和降低功耗最

为迫切的服务器领域。

双核处理器即基于单个半导体的一个处理器上拥有两个一样功能的处理器核心。也就是说，将两个物理处理器核心整合入一个内核中。从理论上讲，由于将两个或多个运算核封装在一个芯片内部，首先节省了大量的晶体管和封装成本（CPU 的核很小，将多个核封装在一起给外形尺寸带来的变化并不显著），同时还能显著提高处理器的性能。另外，由于多核处理器对外的“界面”还是统一的（有的多核产品甚至不会改变引脚数），所以整个计算机系统需要为此做出的改变就很有限，这意味着用户不会在主板、硬件体系方面做大的改变，从兼容性和系统升级成本方面来考虑有诸多的优势。

通常采用两种方式实现两个或多个内核协调工作。一种是采用对称（Symmetric）多处理技术，就像 IBM Power 4 处理器一样，将两颗完全一样的处理器封装在一个芯片内，达到双倍或接近双倍的处理性能，由于共享了缓存和系统总线，因此这种做法的优点是能节省运算资源。另一种技术采用一种非对称多处理（Asymmetric）的工作方式：即两个处理内核彼此不同，各自处理和执行特定的功能，在软件的协调下分担不同的计算任务，比如一个执行加密，而另一个执行 TCP/IP 协议处理。这种处理器的内部结构更像人的大脑，某部分区域（晶体管）在执行某种任务时具有更高的优先级和更强的能力。如 TI 公司的 OMAP5910 双核处理器。

从目前已经发布或透露的多核处理器原型来看，对称式的处理方式将成为未来多核处理器的主要体系结构，同时，多核间将共享大容量的缓存作为处理器之间及处理器与系统内存之间交换数据的“桥梁”。为了提高交换速度，这些缓存往往集成在片内，其数据传输速度是惊人的。

现代的高性能处理器在全速工作时产生的热量是很可观的，某些晶体管在长期高速运算时会因为过热而降低性能，甚至会引起处理器工作状态的不稳定。多核处理器可以有效地避免芯片过热，其解决办法是让两个“核”轮流工作，这样可以解决高频运行中处理器的过热问题。

## 2. 典型多核处理器介绍

下面以 TI 公司的 OMAP 双核处理器 OMAP5910 为例，对双核处理器进行介绍。其功能模块框图如图 2-15 所示。

OMAP（Open Multimedia Applications Platform，开放的多媒体应用平台）是美国德州仪器公司（TI）推出的专门为支持第三代（3G）无线终端应用而设计的应用处理器体系结构。OMAP5910 处理器是由 TI 应用最为广泛的 TMS320C55x DSP 内核与低功耗、增强型 ARM925 微处理器组成的双核应用处理器。TMS320C55x 系列可提供对低功耗应用的实时多媒体处理的支持；ARM925 MPU 可满足控制和接口方面的处理需要。基于双核结构，OMAP5910 同其他 OMAP 处理器一样，采用开放式、易于开发的软件设施，支持广泛的

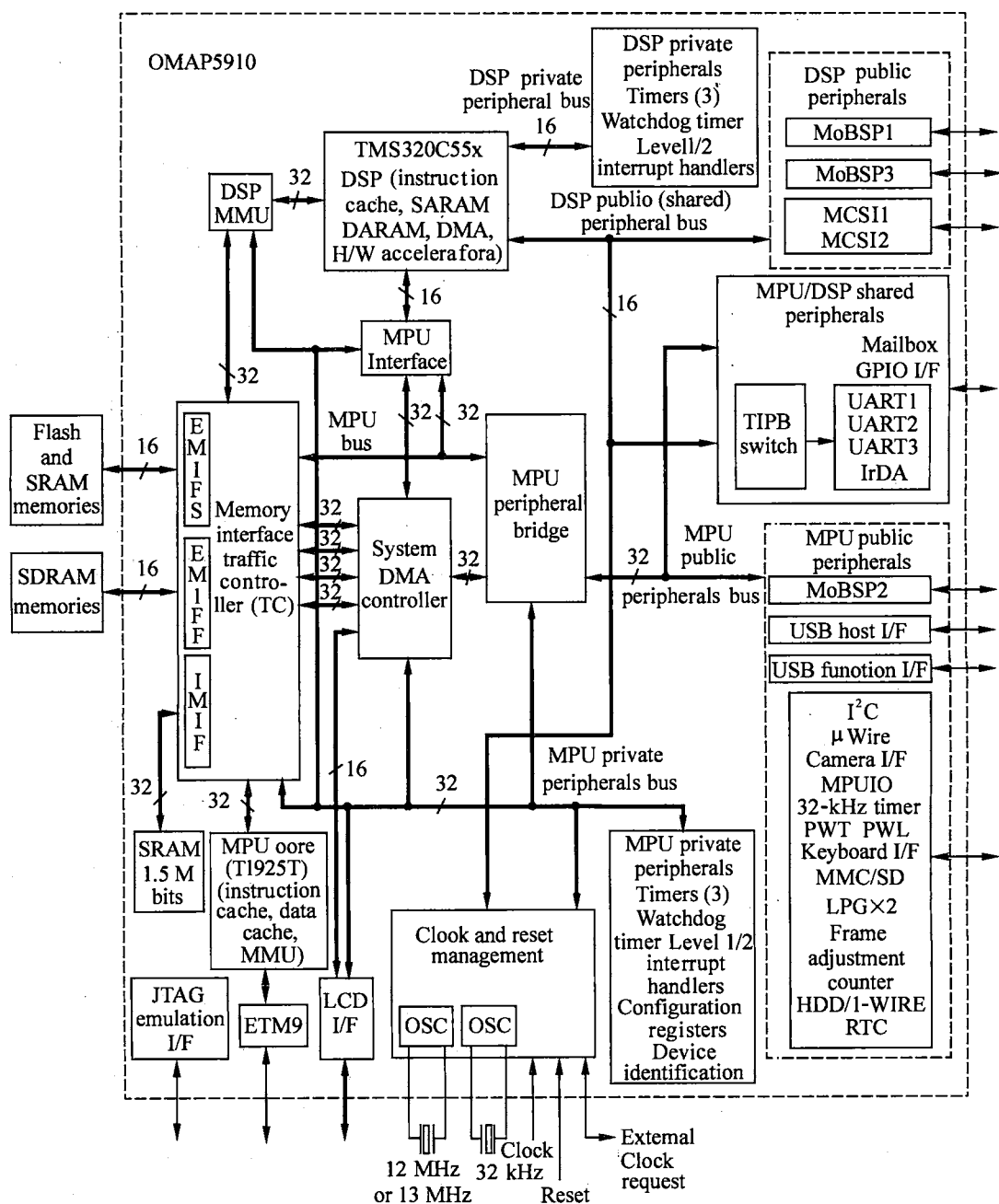


图 2-15 OMAP5910 功能模块框图

操作系统, 如 Linux、Windows、WinCE、Nucleus、Palm OS、VxWorks 等, 优化其应用程序时可以通过 API 及用户熟悉且易于使用的工具。

### (1) OMAP5910 的基本特性

OMAP5910 采用独特的双核结构, 把高性能低功耗的 DSP 核与控制性能强的 ARM 微处理器结合起来, 具有集成度高、硬件可靠性和稳定性强、速度快、数据处理能力强、功耗低、开放性好等优点。OMAP5910 应用处理器双核结构的主要优势在于: 由于两个独立的组件来完成应用处理任务, 其中 MCU 负责支持应用操作系统并完成以控制为核心的应用处理; 而 DSP 则负责完成多媒体信号(如音频、语音和图像/视频信号)的处理。与单核结构相比, 双核体系结构的一个明显优势就是可以使操作系统的效率 and 多媒体代码的执行更加优化并延长电源寿命; 同时采用双处理器可以将总工作负荷合理划分, 从而降低时钟工作频率, 使系统的功耗降低至最低。

### (2) OMAP5910 硬件功能模块

OMAP5910 采用 289 引脚 BGA 封装。其硬件功能模块包括 MCU 子系统、DSP 子系统、存储器管理单元(TC)、直接存储器访问单元(DMA)、两级中断管理器及丰富的外围接口等。其中 DSP 核、ARM 核以及存储器管理单元(TC)这 3 个部分可以独立地进行时钟管理, 从而有效地控制功能。下面简要介绍几个主要功能模块。

#### ① MPU 子系统

MPU 子系统采用 ARM925 核, 5 级流水线结构, 支持 16 位 Thumb 指令集扩展, 工作主频为 175MHz。它包括存储器管理单元、16KB 的高速指令缓冲存储器、8KB 的数据高速缓冲存储器和 17B 的写缓冲器。片内有 192KB 的内部 SRAM, 可为液晶显示器等应用提供大量的数据和代码存储空间。ARM925 核共有 13 个内部中断和 19 个外部中断, 采用两级中断管理。

ARM925 核通过使用协处理器 CP15 使体系结构得到增强。系统中的控制寄存器可通过对协处理器 CP15 的读写来对 MMU、cache 和读写缓存控制器进行存取操作。这种微构架在 ARM 核的周围提供了指令与数据存储器管理单元, 指令、数据和写缓冲器, 性能监控、调试和 JTAG 单元以及协处理器接口, MAC 协处理器和内核存储总线。ARM925 核的 MMU 具有两个 64 项的转换旁路缓存器(TLB)用于指令和数据流, 每项均可映射存储器的段、大页和小页。

#### ② DSP 子系统

DSP 子系统内的 C55x DSP 核具有极佳的功耗性能比, 工作主频为 200MHz。C55x 内核的主要特点是: 有 1 个 64×8 位缓存队列, 2 个 17×17 位乘法器, 1 个 40 位 ALU, 1 个 16 位 ALU, 1 个 40 位桶形移位器和 4 个 40 位加法器。另外还有 12 条独立的总线, 即 3 条数据读总线、2 条数据写总线、5 条数据地址总线、1 条程序读取总线和 1 条程序地址总



线, 以及用户可以配置的 IDLE 域。同时, 内核主要由 4 个单元组成: 指令缓冲单元 (I 单元)、程序流单元 (P 单元)、地址数据流单元 (A 单元) 和数据运算单元 (D 单元)。它支持无线网络传输与语音数据处理等工作, 能提供高效谐振数据处理能力。C55x DSP 核采用了多项新技术: 增大的空闲省电区域、变长指令、扩大的并行机制等。其结构针对多媒体应用做了高度优化, 适合低功耗的实时语音图像处理。C55x DSP 核还新增了图像位移预测、离散余弦变换/反变换和 1/2 像素插值的视频硬件加速器, 从而可以提高数据处理速度, 降低视频处理功耗。此外, 核内还包括 32KB 的双存取 SRAM、48KB 的单存取 SRAM、16KB 的片内 ROM 和 12KB 的高速指令缓存。

### ③ 存储器管理单元 TC

存储器管理单元 TC 管理着 MPU、DSP、DMA 以及局部总线对 OMAP5910 系统存储资源 (如 SRAM、SDRAM、FLASH、ROM 等) 的访问。它的主要功能是确保处理器能够高效访问外部存储区, 并避免产生瓶颈现象而降低片上处理速度。TC 通过 3 种不同的接口支持处理器或 DMA 单元对存储器的访问, 即 EMIFS、EMIFF 和 IMIF。其中 EMIFS 接口提供对 FLASH、SRAM 和 ROM 的访问; EMIFF 接口提供对 SDRAM 的访问; IMIF 接口提供对 OMAP5910 片内 192KB SRAM 的访问。3 个接口是完全独立的, 从任何一个处理器或 DMA 单元都可以同时访问。

### (3) 系统控制功能

OMAP5910 的系统控制模块提供了实时时钟(RTC)、看门狗(WT)、中断控制器、功率管理控制器、复位控制器和两个片上振荡器。

### (4) 时钟和电源管理

OMAP5910 提供了两个振荡器来辅助管理电源耗损, 设计系统时, 在待机模式下可以直接关闭 12MHz 的振荡输入, 只留下 32kHz 振荡器来维持系统运作。

电源管理提供了 3 种工作模式: Awake 模式、Big sleep 模式和 Deep sleep 模式。Awake 模式下, 整个芯片运行在峰值频率, 32kHz 振荡器和 12MHz 振荡器正常工作, 在时钟请求时, 能使能外围器件的 12MHz 时钟, 并由 ULPD DPLL 或 APLL 产生 48MHz 时钟; 当芯片产生 IDLE 请求时, 芯片工作在 Big sleep 模式下, DPLLs 1、内部 12MHz 时钟被关闭; Deep sleep 模式下, 只有 32kHz 振荡器正常工作, 整个系统将处于最低功耗状态。

### (5) 外围控制模块

OMAP5910 微处理器拥有 9 个独立通道和 7 个接收/发送端口的 DMA 控制器。DMA 控制器可响应内部和外部设备的请求, 在 MPU TI925T (ARM9TDMI) 运行的条件下, 完成外部寄存器、内部寄存器和外部设备之间的数据传输。系统 DMA 的设置决定取决于 MPU TI925T (ARM9TDMI) 内核。

OMAP5910 微处理器另外有一个独立 DMA 通道供给 LCD 控制器专用。LCD 控制器

可支持单色和彩色 STN 及彩色 TFT 显示。显示分辨率最大为 1024×1024 像素。在单色模式下, 能支持 15 级灰度; 在 STN 彩色模式下, 最高支持 3375 种颜色; 在 TFT 显示模式下, 最高支持 65536 种颜色。LCD 控制器将帧缓存中的像素编码值, 对应 12 位宽的 256 个入口的调色板 RAM, 根据数据宽度决定彩色的数量。通常可选用片内共享的 SRAM 或者通过 EMIF 接口选用外部 SDRAM 来当作帧缓存器, 为优化性能推荐选用片内共享的 SRAM。

OMAP5910 微处理器支持的串行接口包括: 基于通用串行总线 2.0 版本和开放式主机控制接口 1.0a 版本的 USB 主机功能模块接口; 3 个通用异步收发口 (UART), 其中两个 UART 具有自动调节波特率的性能, 其波特率调节范围在 1200b/s~115.2Kb/s 之间, 而另外一个 UART 通常当作一般的 UART 或者可用作 IrDA 接口使用; 3 个多通道缓冲串行接口 (McBSP), 可提供高达 128 个通道的高速、全双工通信的串行接口, 可直接与 T1/E1 调频器相连接, 并支持兼容 MVIP、ST-BUS、IOM2、AC97、I2S 等协议的设备; 2 个多通道串行接口 (MCSI), 提供了全双工通信以及对主/从时钟的控制功能, 同时, 为 C55x 内核对外部设备诸如多媒体数字音频解码编码器或其他模拟转换器等访问提供便利的通信接口; 基于 Philips I2C 总线 2.1 版本的 I2C 主/从接口, 支持多主机 (Multimaster) 模式, 即在 I2C 总线上的设备 (包括 OMAP5910 在内) 都可充当接收机或发送机; 1 个支持 MMC/SD 或 SPI 协议并传输串行数据的 MMC/SD 卡接口和 1 个 SPI 接口。

#### (6) OMAP5910 的软件构架

OMAP 是一个高度集成的硬件和软件应用平台, 为无线市场提供系统级的解决方案, 从一定意义上来说, 在硬件结构的基础上, OMAP 开放的软件体系结果对用户更为重要。它支持多种流行的嵌入式操作系统、高级语言编程以及丰富的 DSP 多媒体组件算法, 通过应用编程接口 (APIs) 和第三方开发工具, 方便地实现各种应用开发。TI 公司独特的 DSP/BIOS 桥, 允许开发者在 RISC 和 DSP 之间优化的分配任务, 在不增加功耗的前提下获得优化的性能, 从而更好地利用 OMAP 硬件结构的潜能。采用 TI DSP 算法 xDAIS, 可以实现算法的重用, 使已经成熟的 DSP 算法能够快速移植到不同系统, 使开发者能够利用丰富的 DSP 多媒体算法, 快速开发出新产品。

OMAP5910 的软件结构建立在两个操作系统之上: 一是基于 ARM 的 Windows CE、Linux 等操作系统; 二是基于 DSP 的 DSP/BIOS。连接两个操作系统的核心技术是 DSP/BIOS 桥, 它是 OMAP5910 的关键。对于软件开发者来说, DSP/BIOS 桥提供了一种使用 DSP 的无缝接口, 允许开发者在 GPP (通用处理器) 上使用标准应用编程接口访问并控制 DSP 的运行环境。利用 TI 公司的 Code Composer Studio (CCS) 集成开发环境, 从开发者的角度来看, OMAP 好像仅用 GPP 处理器就完成了所有处理功能。这样, 开发者就不需要为两种处理器分别编程, 这使编程工作大为简化。在 OMAP 体系结构下, 开发者可以像对待单个

GPP 那样对 OMAP 的双处理器平台进行编程。而在开发多媒体应用程序时,也可以通过标准的多媒体应用编程接口(MMAPI)使用多媒体引擎,从而方便了应用程序的开发;多媒体引擎对相应的 DSP 任务通过 DSP 应用编程接口(DSPAPI)使用 DSP/BIOS 桥,最后由 DSP/BIOS 桥对数据、I/O 流和 DSP 任务控制进行协调。

## 2.2 嵌入式系统的存储体系

### 2.2.1 存储器系统概述

#### 1. 存储器系统的层次结构

计算机系统的存储器被组织成一个金字塔形的层次结构,如图 2-16 所示。在这个层次结构中,自上而下,依次为 CPU 内部寄存器、芯片内部的高速缓存(cache)、芯片外的高速缓存(SRAM、DRAM、DDRAM)、主存储器(Flash、PROM、EPROM、EEPROM)、外部存储器(磁盘、光盘、CF、SD 卡)和远程二级存储(分布式文件系统、Web 服务器)这 6 个层次的结构。这些设备从上而下,依次变得速度更慢、访问频率更小、容量更大,并且每字节的造价也更加便宜。

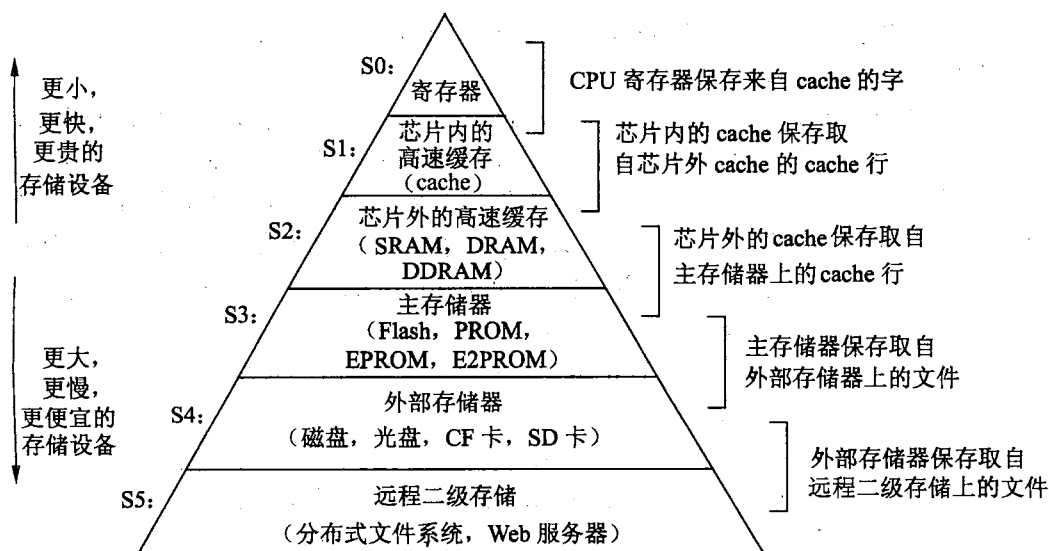


图 2-16 存储器系统层次结构

CPU 内部寄存器位于整个层次结构的最顶部(S0 层),高速缓存(S1 层)保存了 CPU 经常用到的数据,要求在速度上能跟得上 CPU 运算器和控制器的要求,其容量较小,成本

较高。下面依次为内存（S2 层），主存储器（S3 层），外部存储器（S4 层）和远程存储（S5 层）。

在这种存储器分层结构中，上面一层的存储器作为下一层存储器的高速缓存。CPU 寄存器就是 cache 的高速缓存，寄存器保存来自 cache 的字；cache 又是内存层的高速缓存，从内存中提取数据送给 CPU 进行处理，并将 CPU 的处理结果返回到内存中；内存又是主存储器的高速缓存，它将经常用到的数据从 Flash 等主存储器中提取出来，放到内存中，从而加快了 CPU 的运行效率。嵌入式系统的主存储器的容量是有限的，当遇到大信息量的数据时，就需要将其保存到磁盘、光盘或 CF、SD 卡等外部存储器中，并在需要时从外部存储器中提取调用数据。在某些带有分布式文件系统的嵌入式网络系统中，外部存储器就作为其他系统中被存储数据的高速缓存。

## 2. 高速缓存（cache）

为了提高存储器系统的性能，在主存储器和 CPU 之间采用高速缓冲存储器（cache）。cache 被广泛用来提高内存系统性能，许多微处理器体系结构都把它作为其定义的一部分。如果正确使用，cache 能够减少内存平均访问时间。cache 提高了内存访问的可变性，即 cache 中的访问速度最快，而访问不在缓存中的单元会慢一些。

### 1) 高速缓存的分类

#### (1) 统一 cache 和独立的数据/程序 cache

如果一个存储系统中指令预取时使用的 cache 和数据读写时使用的 cache 是用同一个 cache，这时称系统使用统一的 cache。

如果一个存储系统中指令预取时使用的 cache 和数据读写时使用的 cache 是各自独立的，这时称系统使用了独立的 cache。其中，用于指令预取的 cache 称为指令 cache，用于数据读写的 cache 称为数据 cache。

使用独立的数据 cache 和指令 cache，可以在同一个时钟周期中读取指令和数据，而不需要双端口的 cache。但此时要注意保证指令和数据的一致性。

#### (2) 写通 cache 和写回 cache

当 CPU 更新了 cache 的内容时，要将结果写回到主存中，通常有两种方法：写通法（write-through）和写回法（write-back）。

写通法是指 CPU 在执行写操作时，必须把数据同时写入 cache 和主存。采用写通法进行数据更新的 cache 称为写通 cache。

写回法是指 CPU 在执行写操作时，被写的数据只写入 cache，不写入主存。仅当需要替换时，才把已经修改的 cache 块写回到主存中。采用写回法进行数据更新的 cache 称为写回 cache。



### (3) 读操作分配 cache 和写操作分配 cache

当进行数据写操作时，可能 cache 未命中，这时根据 cache 执行的操作不同，将 cache 分为两类：读操作分配 cache 和写操作分配 cache。

对于读操作分配 cache，当进行数据写操作时，如果 cache 未命中，只是简单地将数据写入主存中。主要在数据读取时，才进行 cache 内容预取。

对于写操作分配 cache，当进行数据写操作时，如果 cache 未命中，cache 系统将会进行 cache 内容预取，从主存中将相应的块读取到 cache 中相应的位置，并执行写操作，把数据写入到 cache 中。对于写通类型的 cache，数据将会同时被写入到主存中，对于写回类型的 cache 数据将在合适的时候写回到主存中。

#### 2) cache 的工作原理

在 cache 存储系统当中，把主存储器和 cache 都划分成相同大小的块。主存地址可以由块号 M 和块内地址 N 两部分组成。同样，cache 的地址也由块号 m 和块内地址 n 组成。工作原理图如图 2-17 所示。

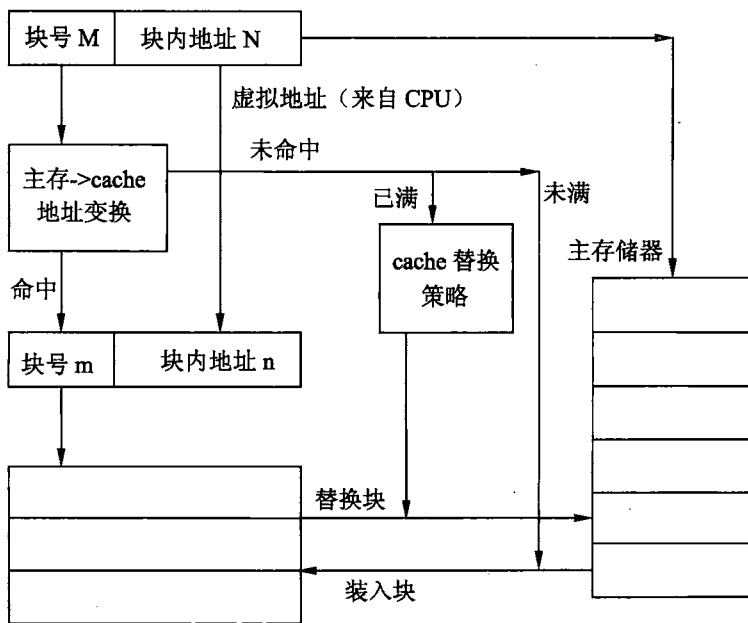


图 2-17 cache 工作原理图

当 CPU 要访问 cache 时，CPU 送来主存地址，放到主存地址寄存器中。然后通过地址变换部件把主存地址中的块号 M 变成 cache 的块号 m，并放到 cache 地址寄存器当中。同时将主存地址中的块内地址 N 直接作为 cache 的块内地址 n 装入到 cache 地址寄存器中。

如果地址变换成功（通常称为 cache 命中），就用得到的 cache 地址去访问 cache，从 cache 中取出数据送到 CPU 中。如果地址变换不成功，则产生 cache 失效信息，并且接着使用主存地址直接去访问主存储器。从主存储器中读出一个字送到 CPU，同时，将从主存储器中读出来的数据装入到 cache 中去。此时，如果 cache 已经满了，则需要采用某种 cache 替换策略（如 FIFO 策略、LRU 策略等）把不常用的块先调出到主存储器中相应的块中，以便腾出空间来存放新调入的块。由于程序具有局部性特点，每次发生失效时都把新的块调入到 cache 中，能够提高 cache 的命中率。

在 cache 当中，地址映像是指把主存地址空间映像到 cache 地址空间。也就是说，把存放在主存中的程序或数据按照某种规则装入到 cache 中，并建立主存地址到 cache 地址之间的对应关系。地址变换是指当程序或数据已经装入到 cache 后，在实际运行过程当中，把主存地址如何变成 cache 地址。

地址映像和地址变换是密切相关的。采用什么样的地址映像就必然会有相应的地址变换与之对应。但是无论采用什么样的地址映像和地址变换方式，都需要把主存和 cache 划分为同样大小的存储单元，通常称存储单元为“块”。在进行地址映像和变换时，都是以块为单位进行调度的。

常用的地址映像和变换方式有：全相联地址映像和变换、组相联地址映像和变换、直接映像和变换。

### （1）直接映像和变换

直接映像不仅快，而且造价相对低。但是由于它将 cache 映射到主存的策略简单，所以有一定的局限性。例如，考虑一个具有 3 个块的直接映射 cache，其中单元 0, 1, 2 分别映射到不同的块。但单元 3, 6, 9 都映射到单元 0 所处的同一块；地址 1, 4, 7, 都映射到单独一块，以此类推。如果访问频繁的块正好被映射到同一个块，就不能充分利用 cache 的好处。

### （2）组相联地址映像和变换

组相联地址映像和变换由它用到的组的个数来标识，给出  $n$  路组相联 cache。每个组被实现为一个直接映射 cache。cache 请求同时广播到所有的组。如果某组中有这个单元，该 cache 便报告命中。虽然内存单元还是以相同的方法映射，但每个单元组都有  $n$  个分离的块。因此，可以将几个恰好映射到相同 cache 块的几个单元同时放入 cache 中。组相联 cache 结构需要一点额外的内部开销并且比直接映射 cache 慢一些，但是它的命中率较高，弥补了其他的不足。

### （3）全相联地址映像和变换

在这种方式当中，主存中任意一个块都可以映射到 cache 中的任意一个块的位置上。

### 3) cache 的替换算法

常用的替换算法有两种：轮转法和随机替换算法。

- 轮转法维护一个逻辑计数器，利用该计数器依次选择将要被替换出去的 cache 块。这种算法容易预测最坏情况下 cache 的性能。当它有一个明显的缺点，在程序发生很小的变化时，可能造成 cache 平均性能急剧的变化。
- 随机替换算法通过一个伪随机数发生器产生一个伪随机数，用新块将编号为该伪随机数的 cache 块替换掉。这种算法很简单，易于实现。但是它没有考虑程序的局部性特点，也没有利用以前块地址分布情况，因而效果较差。同时这种算法不易预测最坏情况下 cache 的性能。

### 3. 存储管理单元

存储管理单元 (Memory Manage Unit, MMU) 在 CPU 和物理内存之间进行地址转换。由于是将地址从逻辑空间映射到物理空间，因此这个转换过程一般称为内存映射。MMU 主要完成以下工作：

- 虚拟存储空间到物理存储空间的映射。采用了页式虚拟存储管理，它把虚拟地址空间分成一个个固定大小的块，每一块称为一页，把物理内存的地址空间也分成同样大小的页。MMU 实现的就是从虚拟地址到物理地址的转换。
- 存储器访问权限的控制。
- 设置虚拟存储空间的缓冲的特性。

嵌入式系统中常常采用页式存储管理。为了管理这些页引入了页表的概念。页表是位于内存中的表，它的每一行对应于虚拟存储空间的一个页，该行包含了该虚拟内存页对应的物理内存页的地址、该页的方位权限和该页的缓冲特性等。在基于 ARM 的嵌入式系统中，使用系统控制协处理器 CP15 的寄存器 C2 来保存页表的基地址。

从虚拟地址到物理地址的变换过程就是查询页表的过程。由于页表是存储在内存中的，整个查询过程需要付出很大的代价。基于程序在执行过程中具有局部性的原理，在一段时间内，对页表的访问只是局限在少数几个单元。根据这一特点，增加了一个小容量（通常为 8-16 字）、高速度（访问速度和 CPU 中通用寄存器相当）的存储部件来存放当前访问需要的地址变换条目，这个存储部件称为地址转换后备缓冲器 (Translation Lookaside Buffer, TLB)。

当 CPU 访问内存时，首先在 TLB 中查找需要的地址变换条目，如果该条目不存在，CPU 在从位于内存中的页表中查询，并把相应的结果添加到 TLB 中，更新它的内容。这样做的好处是，如果 CPU 下一次又需要该地址变换条目时，可以从 TLB 中直接得到，从而使地址变换的速度大大加快。当内存中的页表内容改变，或者通过修改系统控制协处理器 CP15 的寄存器 C2 使用新的页表时，TLB 中的内容需要全部的清除。MMU 提供了相关

的硬件支持这种操作。系统控制协处理器 CP15 的寄存器 C8 用来控制清除 TLB 内容的相关操作。

MMU 可以将某些地址变换条目锁定在 TLB 中,从而使得进行与该地址变换条目相关的地址变换速度保持很快。在 MMU 中寄存器 C10 用于控制 TBL 内容的锁定。MMU 可以将整个存储空间分成最多 16 个域,每个域对应一定的内存区域,该区域具有相同的访问控制属性。MMU 中寄存器 C3 用于控制与域相关的属性的配置。当存储访问失效时,MMU 提供了相应的机制来处理这种情况。MMU 中寄存器 C5 和寄存器 C6 用于支持这些机制。

下面列出了与 MMU 操作相关的寄存器,如表 2-6 所示。

表 2-6 与 MMU 操作相关的寄存器

寄 存 器	作 用	寄 存 器	作 用
寄存器 C1	配置 MMU 中一些操作	寄存器 C5	内存访问失效状态指示
寄存器 C2	保存内存中页表的基地址	寄存器 C6	内存访问失效时失效的地址
寄存器 C3	设置域的访问控制属性	寄存器 C8	控制与清除 TLB 内容相关操作
寄存器 C4	保留	寄存器 C10	控制与锁定 TLB 内容相关操作

### 1) MMU 的存储访问过程

CP15 的寄存器 C1 的位[0]用于控制禁止/使能 MMU。当 CP15 的寄存器 C1[0]=0 时,禁止 MMU;当 C1[0]=1 时,使能 MMU。下面给出了禁止 MMU 的指令:

```
MRC P15,0,R0,C1,0,0
```

```
ORR R0,0
```

```
MCR P15,0,R0,C1,0,0
```

### (1) 使能 MMU 时存储访问过程

当 ARM 处理器请求存储访问时,首先在 TLB 中查找虚拟地址。如果系统中数据 TLB 和指令 TLB 是分开的,在取指令时,从指令 TLB 查找相应的虚拟地址,对于内存访问操作,从数据 TLB 中查找相应的虚拟地址。

在这个过程当中,如果该虚拟地址对应的地址变换条目不在 TLB 中,CPU 从位于内存中的页表中查询对应于该虚拟地址的地址变换条目,并把相应的结果添加到 TLB 中。如果 TLB 已经满了,则需要根据一定的替换算法进行替换。这样当 CPU 下次再次访问时,可以从 TLB 中直接得到,从而使地址变换的速度大大加快。

当得到了需要的地址变换条目后,将进行以下的操作:

① 得到该虚拟地址对应的物理地址。

② 根据条目中 C (cachable) 控制位和 B (Bufferable) 控制位决定是否缓存该内存访问的结果。

③ 根据存储权限控制位和域访问控制位确定该内存访问是否被允许。如果该内存访问不被允许，CP15 向 ARM 处理器报告存储访问中止。

④ 对于不允许缓存 (uncached) 的存储访问，使用步骤①中得到的物理地址访问内存。对于允许缓存 (cached) 的存储访问，如果在 cache 命中，则忽略物理地址；如果 cache 没有命中，则使用步骤①中得到的物理地址访问内存，并把该块数据读取到 cache 中。

允许缓存的 MMU 存储访问示意图如图 2-18 所示。

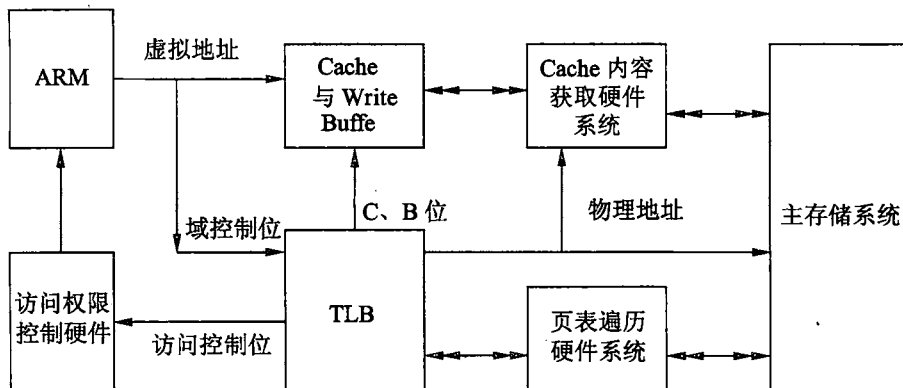


图 2-18 允许缓存的 MMU 存储访问示意图

## (2) 禁止 MMU 时存储访问过程

当禁止 MMU 时，按照如下的存储访问规则：

- 当禁止 MMU 时，先要确定芯片是否支持 cache 和 write buffer。如果芯片规定当禁止 MMU 时禁止 cache 和 write buffer，则存储访问将不考虑 C 和 B 控制位。如果芯片规定当禁止 MMU 时可以使能 cache 和 write buffer，则数据访问时，C=0,B=0；指令读取时，如果使用分开的 TLB，那么 C=1，如果使用统一的 TLB，那么 C=0。
- 存储访问不进行权限控制，MMU 也不会产生存储访问中止信号。
- 所有的物理地址和虚拟地址相等，即使用平板存储模式。

## 2) MMU 中的地址变换过程

嵌入式系统中虚拟存储空间到物理存储空间的映射以内存块为单位来进行。即虚拟存储空间中一块连续的存储空间被映射到物理存储空间中同样大小的一块连续存储空间。在页表和 TLB 中，每一个地址变换条目实际上记录了一个虚拟存储空间的内存块的基地址与物理存储空间相应的一个内存块的基地址的对应关系。根据内存块大小，可以有多种地址变换。

嵌入式系统支持的内存块大小有以下几种：

- 段 (section) 大小为 1MB 的内存块。
- 大页 (Large Pages) 大小为 64KB 的内存块。
- 小页 (Small Pages) 大小为 4KB 的内存块。
- 极小页 (Tiny Pages) 大小为 1KB 的内存块。

极小页只能以 1KB 大小为单位不能再细分,而大页和小页有些情况下可以在进一步的划分,大页可以分成大小为 16KB 的子页,小页可以分成大小为 1KB 的子页。

在 MMU 中实现虚拟地址到物理地址的映射是通过两级页表来实现的:

- 一级页表中包含有以段为单位的地址变换条目及指向二级页表的指针。一级页表是实现地址映射粒度较大。以段为单位的地址变换过程只需要一级页表。
- 二级页表中包含有以大页和小页为单位的地址变换条目。有一种类型的二级页表还包含有以极小页为单位的地址变换条目。以页为单位的地址变换过程需要二级页表。

### 3) MMU 中的存储访问权限控制

在 MMU 中,寄存器 C1 的 R、S 控制位和页表中地址转换条目中的访问权限控制位联合作用,以控制存储访问的权限。MMU 存储访问权限控制的具体规则如表 2-7 所示。

表 2-7 MMU 中存储访问权限控制

AP	SR	特权级时访问权限	用户级时访问权限
0b00	00	没有访问权限	没有访问权限
0b00	10	只读	没有访问权限
0b00	01	只读	只读
0b00	11	不可预知	不可预知
0b01	XX	读/写	没有访问权限
0b10	XX	读/写	只读
0b11	XX	读/写	读/写

### 4) MMU 中的域

MMU 中的域指的是一些段、大页或者小页的集合。每个域的访问控制特性都是由 CP15 中的寄存器 C3 中的两位来控制的。因此就能很容易地将某个域的地址空间包含在虚拟存储空间中,或是排除在虚拟存储空间之外。

CP15 中寄存器 C3 中的每两位控制一个域的访问控制特性,字段编码和含义如表 2-8 所示。

### 5) 快速上下文切换技术

快速上下文切换技术 (Fast Context Switch Extension, FCSE) 通过修改系统中不同进

程的虚拟地址，避免在进行进程间切换时造成的虚拟地址到物理地址的重映射，从而提高系统的性能。

表 2-8 字段编码和含义

控制位编码	方位类型	含 义
0b00	没有访问权限	这时访问该域将产生访问失效
0b01	客户类型	根据页表中地址变换条目中的访问权限控制位决定是否运行特定的存储访问
0b10	保留	使用该值会产生不可预知的结果
0b11	管理者权限	不考虑页表中地址变换条目中的访问权限控制位，这种情况下不会产生访问失败

如果两个进程占用的虚拟地址空间有重叠，则系统在这两个进程之间进行切换时，必须进行虚拟地址到物理地址的重映射，包括重建 TLB、清除 cache，整个工作需要巨大的系统开销，而快速上下文切换技术的引入避免了这种开销。

FCSE 位于 CPU 和 MMU 之间，其责任就是将不同进程使用的相同虚拟地址映射为不同的虚拟空间，使得在上下文切换时无需重建 TLB 等。

如果两个进程使用了同样的虚拟地址空间，则对 CPU 而言，FCSE 对各个进程的虚拟地址进行变换，这样系统中除了 CPU 之外的部分，看到的是经过快速上下文切换技术变换后的虚拟地址。

#### 6) 存储器映射的输入/输出

在嵌入式系统中，I/O 操作通常被映射成存储器操作，即输入/输出是通过存储器映射的可寻址外围寄存器和中断输入的組合来实现的。

在嵌入式中，I/O 的输出操作可通过存储器写入操作实现；I/O 的输入操作可通过存储器读取操作实现。这样 I/O 空间就被映射成存储空间。但这些存储器映射的 I/O 空间不满足 cache 所要求的特性。例如，从一个普通的存储单元连续读取两次，将会返回同样的结果。对于存储器映射的 I/O 空间，连续读取两次返回的结果可能不同。这可能是由于第一次读操作有副作用，或者其他操作可能影响了该存储器映射的 I/O 单元的内容，因此，对于存储器映射的 I/O 空间的操作，就不能使用 cache 技术。某些 ARM 系统也可能有存储器直接访问（DMA）硬件。

### 2.2.2 嵌入式系统存储设备分类

存储器就是用来存储信息的部件，因为有了存储器，嵌入式系统才有了对信息的记忆功能。存储器是嵌入式系统硬件的重要组成部分，用来存放嵌入式系统工作时所用信息——

程序和数据。

在设计嵌入式系统的存储器时需要考虑许多因素：有的嵌入式处理器集成了存储器，一般不需要扩展；有的没有集成存储器，就必须扩展；而有的处理器虽然集成了一定数量的存储器，可以满足一定的需要，但是由于应用软件比较大，需要扩展处理器。整个嵌入式系统的存储器由片内和片外两部分组成。

### 1. 存储器部件的分类

#### (1) 按在系统中的地位分类

根据存储器在微机系统中的不同地位，可分为主存储器（Main Memory，简称内存或主存）和辅助存储器（Auxiliary Memory，Secondary Memory，简称辅存或外存）。内存是计算机主机的一个组成部分，用来容纳当前正在使用的或要经常使用的程序和数据，对于内存，CPU 可以直接对它进行访问。外存也是用来存储各种信息的，但是 CPU 要使用这些信息时，必须通过专门的设备将信息先传送到内存中。因此，外存放的是相对来说不经常使用的程序和数据，另外，外存总是和某个外部设备相关的。

因为内存可由 CPU 直接存取，再加上一般都用快速存储器件来构成内存，这就使内存的存取速度很快。但是，内存空间的大小受到地址总线位数的限制。例如，在 8086/8088 微型计算机系统中，地址总线是 20 位，所以最大的内存空间为  $2^{20} \text{B} = 1\text{MB}$ 。正是内存的快速存取和容量受限制的特点，使它主要用来存放系统软件、参数及当前要运行的应用程序和数据。系统软件中有一部分软件如引导程序、监控程序或者操作系统中的基本输入/输出部分 BIOS，都是无时无刻不用的，它们必须常驻内存；更多的系统软件和全部应用软件则在用到时由外存传送到内存。

作为一个嵌入式计算机系统，在长期使用中，必须有许多程序和数据要存储起来，因此，只有内存就不够了，另外，在实际应用中，希望既能方便地对程序进行修改，又能对它作长期保存，而这也是当前大多数构成内存的器件所不能实现的功能。于是，人们又设计出各种外部存储器。当前，在微型计算机中常见的外存有软盘、硬盘、U 盘等，近年来，光盘也逐渐进入使用阶段。外部存储器的容量不受限制。不过，外存都要配置专门的驱动设备才能完成访问功能。比如，软盘要配置软盘驱动器，硬盘要配置硬盘驱动器。所以，外存的特点是大容量，所存信息既可修改又可保存，但速度比较慢，要配置专用设备。

#### (2) 按存储介质分类

根据存储介质的材料及器件的不同，可分为磁存储器（Magnetic Memory），半导体集成电路存储器（通常称为半导体存储器）、光存储器（Optical Memory）及激光光盘存储器（Laser Optical Disk）。其中磁存储器中又分为磁芯（Magnetic Core）、磁泡（Magnetic Bubble）、磁鼓（Magnetic Drum）、磁带（Magnetic Tape）和磁盘（Magnetic Disk）等。

#### (3) 按信息存取方式分类





存储器按存储信息的功能,分为随机存取存储器(Random Access Memory, RAM)和只读存储器(Read Only Memory, ROM)。随机存取存储器又称读写存储器,一般是指机器运行期间可读、可写的存储器。而只读存储器一般是指机器运行期间只能读出信息,不能随时写入信息的存储器。然而实际上所谓随机存取即指随意存取,是相对于顺序存取而言的。对随机存取的存储器来说,当要取出某一单元的信息时,无需经过中间单元而耗费不必要的时间,即随机存取能做到信息的存取时间与其所在的位置无关。

只读存储器按功能可分为掩模式(简称 ROM)、可编程只读存储器(Programmable ROM, PROM)和可改写的只读存储器(Erasable Programmable ROM, EPROM)3种。随机存储器按信息存储的方式,可分为静态 RAM(Static RAM, 简称 SRAM), 动态 RAM(Dynamic RAM, 简称 DRAM)及准静态 RAM(Pseudostatic RAM, 简称 PSRAM)3种。

## 2. 存储器的组织和结构

描述存储器的最基本的参数是存储器的容量,如 4MB。通常,存储器的表示并不唯一,有一些不同表示方法,每种有不同的数据宽度。例如,一个 4MB 的存储器可能有下列两种表示:

- 一个  $1\text{ M} \times 4$  位的阵列,每次存储器访问可获得 4 位数据项,最大共有 220 个不同地址。
- 一个  $4\text{ M} \times 1$  位的阵列,每次存储器访问可获得 1 位数据项,最大共有 222 个不同地址。

在存储器内部,数据是存放在二维阵列存储单元中。因为阵列以二维的形式存储,所以给出的  $n$  位地址被分成行地址和列地址( $n = r + c$ )。 $r$  是行地址数, $c$  是列地址数。行列选定一个特定存储单元。如果存储器外部宽度为 1 位,那么列地址仅一位;对更宽的数据,列地址可选择所有列的一个子集。

嵌入式系统属于专用的系统,受到体积、功耗和成本等各方面因素的影响,因此,它的存储器与通用系统的存储器有所不同。嵌入式存储器一般采用存储密度较大的存储器芯片,存储容量与应用的软件大小相匹配,有时为了设计的需要还要求能够扩展存储器系统。典型的嵌入式存储器系统由 ROM、RAM、EPROM 等组成的。如图 2-19 所示为嵌入式系统的存储器空间分配示意图:



图 2-19 典型的嵌入式系统存储空间分配

## 3. 常见的嵌入式系统存储设备

### (1) RAM(随机存储器)

RAM 可以被读和写。它与磁盘不同,地址可以以任意次序被读。RAM 可以分为:SRAM(静态随机存储器)和 DRAM(动态随机存储器)。这两类存储器具有不同的特征:

- SRAM 比 DRAM 运行速度快。

- SRAM 比 DRAM 耗电多。
- 在一个芯片上可以置放更多的 DRAM。
- DRAM 需要周期性刷新。

## (2) ROM (只读存储器)

ROM 用固定数据预编程。它在嵌入式系统中非常有用, 因为许多代码或数据不随时间改变。只读存储器对辐射感应的错误也相对不敏感。

可用的只读存储器通常有工厂编程的只读存储器 (有时被称为掩模编程只读存储器 (mask-programmed ROM)) 和现场可编程只读存储器。前者从工厂订购时已写入特定程序。它们一般被成千上万地大量订购, 但很明显, 只有当 ROM 以一定数量安装时工厂编程才有用。后者可在实验室内被编程。编程单元有时被称为 ROM 编程器。为给 ROM 编程, 用户以标准格式生成一编程文件, 将 ROM 插进 ROM 编程器, 发送文件到编程器给 ROM 编程。

## 2.2.3 ROM 的种类与选型

### 1. 常见 ROM 的种类 (PROM、EPROM、E2PROM)

ROM 的特点为在烧入数据后, 无需外加电源来保存数据。断电数据不丢失, 但速度较慢, 因此适合存储需长期保留的不变数据。常见 ROM 的分类如下。

- Mask ROM (掩模 ROM): 一次性由厂家写入数据的 ROM, 用户无法修改。
- PROM (Programmable ROM 可编程 ROM): 和掩模 ROM 不同的是出厂时厂家并没有写入数据, 而是保留里面的内容为全 0 或全 1, 由用户来编程一次性写入数据, 也就是改变部分数据为 1 或 0。
- EPROM (Erasable Programmable ROM 电可擦写 ROM): EPROM 是通过紫外光的照射, 擦掉原先的程序。芯片可重复擦除和写入, 解决了 PROM 芯片只能写入一次的弊端。
- EEPROM (E2PROM) 电可擦除可编程 ROM: EEPROM 是通过加电擦除原数据, 通过高压脉冲可以写入数据。使用方便但价格较高, 而且写入时间较长, 写入较慢。
- Flash ROM (闪速存储器): Flash ROM 具有结构简单、控制灵活、编程可靠、加电擦写快捷的优点, 而且集成度可以做得很高, 它综合了前面的所有优点: 不会断电丢失数据 (NVRAM), 快速读取, 电可擦写可编程 (EEPROM), 因此在手机, PC, PPC 等电器中成功地获得了广泛的应用。

### 2. PROM、EPROM、E2PROM 型 ROM 的各自典型特征和不同点

#### (1) PROM

PROM (Programmable Read-Only Memory, 可编程只读存储器) 因为只允许用户利

用专门的设备（编程器）将自己的程序写入一次，一旦写入后，其内容将无法改变，所以也称一次可编程只读存储器（One Time Programming ROM）。

PROM 产品出厂时，所有记忆单元均制成“0”（或制成“1”），用户可以根据需要将其中的某些单元写入数据 0 或 1，以实现对其“编程”的目的。PROM 根据写入原理可分为两类：结破坏型和熔丝型。由于它们的写入都是不可逆的，所以只能进行一次性写入。

结破坏型在每个行、列线的交点处，制造一对彼此反向的二极管，它们因为彼此反向而不能导通，故全部为“0”。若某位需要写入“1”，则在相应的行、列之间加上较高电压，将反偏的一只二极管永久性击穿，只留下正向导通的一只二极管，故该位被写入“1”。显然这种写入是一次性的，不可逆转的。

熔丝型的基本记忆单元电路是由三极管 T 连接一段镍—铬熔丝组成的，如图 2-20 所示。典型的 PROM 芯片出厂时，T 与位线之间的熔丝都存在，表示全部内容均为“0”。当用户需要在某一位写入“1”时，设法将 T 管的电流加大为正常工作电流的 5 倍以上，从而使镍—铬熔丝熔断，“1”被写入。由于熔丝熔断之后不能再恢复，这种写入也是不可逆转的。

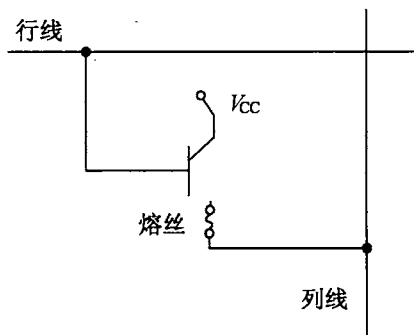


图 2-20 熔丝型 PROM

## （2）EPROM

EPROM（Erasable Programmable Read-Only Memory，可擦除可编程只读存储器）不仅可以由用户利用编程器写入信息，而且可以对其内容进行多次改写。

EPROM 出厂时，存储内容为全“1”，用户可以根据需要将其某些记忆单元改为“0”。当需要更新存储内容时可以将原存储内容擦除（恢复全“1”），以便再写入新的内容。

EPROM 又可分为两种：UVEPROM（紫外线擦除）和 EEPROM（电擦除）。

UVEPROM 需用紫外线灯制作的擦抹器照射存储器芯片上的透明窗口，使芯片中原存内容被擦除。由于是用紫外线灯进行擦除，所以只能对整个芯片擦除，而不能对芯片中个

别需要改的存储单元单独擦除。另外,为了防止存储的信息受日光紫外线成分的作用而缓慢丢失,在 UVEPROM 芯片写入完成后,必须用不透明的黑纸将芯片上的透明窗口封住。

EEPROM 是采用电气方法来进行擦除的,在联机条件下既可以用字擦除,也可以用数据块擦除方式擦除。以字擦除方式操作时,能够只擦除被选中存储单元的内容;以数据块擦除方式操作时,可擦除数据块内所有单元的内容。

EPROM 虽然既可读,又可“写”,但它却不能取代 RAM。因为 EPROM 的编程次数(寿命)是有限的;而且它的写入时间过长,即使对于 EEPROM,擦除一个字节需要约 10ms,写入一个字节大约需要 10 $\mu$ s,比 SRAM 或 DRAM 的时间长 100~1000 倍。

### (3) EEPROM

为了省却消除 EPROM 上数据需照射紫外线的麻烦程序,在 20 世纪 80 年代所开发的电擦除式 PROM (Electrically Erasable Programmable Read-Only Memory, EEPROM) 就很受开发人员的欢迎,EEPROM 不但可以利用电压的高低来写入数据,还可以直接利用电压的高低清除 EEPROM 所存储的数据。

EEPROM 的工作原理:EEPROM 要写入数据“0”的时候,栅极也接上高电压(通常为 17V),P 型基底接上 0V 的电压,源极接上高电压,漏极也接上高电压,浮动栅极则被注入电子。EEPROM 要写入数据“1”的时候,栅极接上高电压(通常 17V),P 型基底接上 0V 的电压,源极接上 0V,漏极接上高电压,浮动栅极保持原本清除状态。

EEPROM 数据写入后,源极的电压接上高电压,表示 EEPROM 在内存的保持状态,要读取数据的时候,源极电压接到 0V,源-漏极间会根据浮动栅极是否有足够电子在来接通,如果浮动栅极有足够的电子,则源-漏极为开路,表示数据为“1”,若浮动栅极没有足够的电子,则源-漏极为短路,表示数据为“0”。

当要清除 EEPROM 上的数据,就将栅极电压接到 0V,P 基底接上高电压,源极与漏极接上高电压,浮动栅极的电子会穿越绝缘体往 P 基底放电,完成 EEPROM 的数据清除工作。

EEPROM 在数据清除的时候可以针对个别的存储单元进行清除操作,比起 EPROM 需整个清除数据方便许多。EEPROM 的数据存储保持能力可以长达 10 年,而数据清除再被规划的次数可以达到一万次以上,因此 EEPROM 的使用比 EPROM 更为普遍。

目前常用的 EEPROM 有 Intel 公司的 28 系列、Winbond 公司的 W27-29 系列及 AMD 公司的 29 系列等,其中 Intel 公司的 28 系列 EEPROM 上的编号代表意义与 EPROM 相似,内容大同小异。

## 2.2.4 Flash Memory 的种类与选型

Flash memory 是嵌入式系统中重要的组成部分,它在嵌入式系统中的功能可以和硬盘

在 PC 中的功能相比，它们都是用来存储程序和数据的，而且可以在掉电情况下继续保存数据使其不会丢失。

### 1. Flash Memory 的种类（NOR 和 NAND 型）

Flash Memory（闪存存储器）作为一种安全、快速的存储体，具有体积小、容量大、成本低、掉电数据不丢失等一系列优点，已成为嵌入式系统中数据和程序最主要的载体。由于 Flash Memory 在结构和操作方式上与硬盘、E<sup>2</sup>ROM 等其他存储介质有较大区别，使用 Flash Memory 时必须根据其自身特性，对存储系统进行特殊设计，以保证系统的性能达到最优。

Flash Memory 是一种非易失性存储器 NVM（Non-Volatile Memory），根据结构的不同可以将其分成 NOR Flash 和 NAND Flash 两种。Flash Memory 具有如下特点：

- 区块结构

Flash Memory 在物理结构上分成若干个区块，区块之间相互独立。比如 NOR Flash 把整个存储区分成若干个扇区（Sector），而 NAND Flash 把整个存储区分成若干个块（Block）；

- 先擦后写

由于 Flash Memory 的写操作只能将数据位从 1 写成 0，不能从 0 写成 1，所以在对存储器进行写入之前必须先执行擦除操作，将预写入的数据位初始化为 1。擦操作的最小单位是一个区块，而不是单个字节。

- 操作指令

除了 NOR Flash 的读，Flash Memory 的其他操作不能像 RAM 那样，直接对目标地址进行总线操作。比如执行一次写操作，它必须输入一串特殊的指令（NOR Flash），或者完成一段时序（NAND Flash）才能将数据写入到 Flash Memory 中。

- 位反转

由于 Flash Memory 固有的电器特性，在读写数据过程中，偶尔会产生一位或几位数据错误。这就是位反转。位反转无法避免，只能通过其他手段对结果进行事后处理。

- 坏块

Flash Memory 在使用过程中，可能导致某些区块的损坏。区块一旦损坏，将无法进行修复。如果对已损坏的区块进行操作，可能会带来不可预测的错误。尤其是 NAND Flash 在出厂时就可能存在这样的坏块（已经被标识出）。

NOR Flash 的特点是应用程序可以直接在闪存内运行，不需要再把代码读到系统 RAM 中运行。NOR Flash 的传输效率很高，在 1~4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND Flash 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快，这也是为何所有的 U 盘都使用 NAND Flash 作为存储介质的原因。应用 NAND Flash 的困难在于闪存和需要特殊的系统接口。

## 2. NOR 和 NAND 型 Flash Memory 各自的典型特征和不同点

### (1) 性能比较

Flash Memory 是非易失内存, 可以对以块或扇区为单位的内存单元进行擦写和再编程。任何闪存器件的写入操作只能在空或已擦除的单元内进行, 所以大多数情况下, 在进行写操作之前必须先执行擦除。NAND Flash 执行擦除操作是十分简单的, 而 NOR 型内存则要求在进行擦除前先要将目标块内所有的位都写为 0。

由于擦除 NOR 型内存时是以 64~128KB 为单位的块进行的, 执行一个写入/擦除操作的时间为 5s, 与此相反, 擦除 NAND Flash 是以 8~32KB 的块进行的, 执行相同的操作最多只需要 4ms。

执行擦除时块尺寸的不同进一步拉大了 NOR 和 NAND Flash 之间的性能差距, 统计表明, 对于给定的一套写入操作 (尤其是更新小文件时), 更多的擦除操作必须在基于 NOR Flash 的单元中进行。这样, 当选择存储解决方案时, 设计师必须权衡以下的各项因素:

- NOR Flash 的读速度比 NAND Flash 稍快一些。
- NAND Flash 的写入速度比 NOR Flash 快很多。
- NAND Flash 的 4ms 擦除速度远比 NOR Flash 的 5s 快。大多数写入操作需要先进行擦除操作。
- NAND Flash 的随机读取能力差, 适合大量数据的连续读取。

### (2) 接口差别

NOR Flash 带有 SRAM 接口, 有足够的地址引脚来寻址, 可以很容易地存取其内部的每一个字节。NAND Flash 地址、数据和命令共用 8 位总线 (Samsung 公司某些新的 NAND Flash 有 16 位总线), 每次读写都要使用复杂的 I/O 接口串行地存取数据, 8 个引脚用来传送控制、地址和资料信息。

NAND Flash 读和写操作采用 512B 的块, 有点像硬盘管理操作。因此, 基于 NAND 的闪存可以取代硬盘或其他块设备。

### (3) 容量和成本

NAND Flash 的单元尺寸几乎是 NOR Flash 的一半, 由于生产过程更为简单, NAND Flash 结构可以在给定的模具尺寸内提供更高的容量, 也就相应地降低了价格。

NOR Flash 容量一般较小, 通常在 1MB~8MB 之间。而 NAND Flash 只是用在 8MB 以上的产品当中, 这也说明 NOR Flash 主要应用在代码存储介质中, NAND Flash 适用于资料存储。NAND Flash 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

### (4) 可靠性和耐用性

采用 Flash Memory 介质时一个需要重点考虑的问题是可靠性。对于需要扩展 MTBF



的系统来说, Flash Memory 是非常合适的存储方案。可以从寿命(耐用性)、位交换和坏块处理三个方面来比较 NOR Flash 和 NAND Flash 的可靠性。

- 寿命(耐用性):在 NAND Flash 中每个块的最大擦写次数是一百万次,而 NOR Flash 的擦写次数是十万次。NAND Flash 除了具有 10:1 的块擦除周期优势,典型的 NAND Flash 块尺寸要比 NOR 型闪存小 8 倍,每个 NAND Flash 的内存块在给定的时间内删除次数要少一些。
- 位交换:所有 Flash Memory 器件都受位交换现象的困扰。在某些情况下(很少见, NAND Flash 发生的次数要比 NOR Flash 多),一个比特位会发生反转或被报告反转了。一位的变化可能不很明显,但是如果发生在一个关键文件上,这个小小的故障可能导致系统停机。如果只是报告有问题,多读几次就可能解决了。当然,如果这个位真的改变了,就必须采用错误探测/错误纠正(EDC/ECC)算法。位反转的问题更多见于 NAND Flash, NAND Flash 的供货商建议使用 NAND Flash 的时候,同时使用 EDC/ECC 算法。这个问题对于用 NAND 存储多媒体信息时倒不是致命的。当然,如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时,必须使用 EDC/ECC 系统以确保可靠性。
- 坏块处理: NAND Flash 中的坏块是随机分布的。以前也曾有过消除坏块的努力,但发现成品率太低,代价太高,根本不划算。NAND Flash 需要对介质进行初始化扫描以发现坏块,并将坏块标记为不可用。在已制成的器件中,如果通过可靠的方法不能进行这项处理,将导致高故障率。

#### (5) 易用性

可以非常直接地使用基于 NOR Flash, 可以像其他内存那样连接,并可以在上面直接运行代码。

由于需要 I/O 接口, NAND Flash 要复杂得多。各种 NAND Flash 的存取方法因厂家而异。

在使用 NAND Flash 时,必须先写入驱动程序,才能继续执行其他操作。向 NAND Flash 写入信息需要相当的技巧,因为设计师绝不能向坏块写入,这就意味着在 NAND Flash 上自始至终都必须进行虚拟映像。

#### (6) 软件支持

当讨论软件支持的时候,应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和 Flash Memory 管理算法的软件,包括性能优化。

在 NOR Flash 上运行代码不需要任何的软件支持。位在 NAND Flash 上进行同样操作时,通常需要驱动程序,也就是内存技术驱动程序(MTD), NAND Flash 和 NOR Flash 在

进行写入和擦除操作时都需要 MTD。

### (7) 市场定位

根据前面所介绍的 NAND Flash 和 NOR Flash 的特点, 两者各自拥有相应不同的应用。一般说来, NOR Flash 用于对数据可靠性要求较高的代码存储、通信产品、网络处理等领域; 而 NAND Flash 则用于对存储容量要求较高的 MP3、存储卡、U 盘等领域。正是如此, NOR Flash 也被称为代码闪存 (Code Flash), 而 NAND Flash 也被称为数据闪存 (Data Flash)。

## 2.2.5 RAM 的种类与选型

### 1. 常见 RAM 的种类 (SRAM、DRAM、DDRAM)

常见 RAM 的种类有 SRAM (Static RAM, 静态随机存储器)、DRAM (Dynamic RAM, 动态随机存储器)、DDRAM (Double Data Rate SDRAM, 双倍速率随机存储器)。

SRAM 采用了与制作 CPU 相同的半导体工艺, 因此与 DRAM 比较, SRAM 的存取速度快, 但制造成本高。DRAM 的应用较广, 经常用做嵌入式计算机系统内存使用。DDRAM 是基于 SDRAM 的一种新技术, 是 SDRAM 的下一代产品, 它在本质上和 SDRAM 完全相同。两者的最大区别在于 DDRAM 采用的新内存模块的时钟频率与普通 SDRAM 的速度一样时, 它可以通过在同一时钟周期的上升和下降沿中都传送数据, 使得 DDRAM 内存比普通 SDRAM 的带宽提升了一倍, 也即是说在同样的时间内传送的数据量增加了一倍。

### 2. SRAM、DRAM、DDRAM 型 RAM 的各自典型特征和不同点

#### (1) SRAM

SRAM 是静态的, 因此只要供电它就会保持一个值。SRAM 没有刷新周期, 由触发器构成基本单元, 集成度低, 每个 SRAM 存储单元由 6 个晶体管组成, 因此其成本较高。SRAM 具有较高的速率, 常常用于高速缓冲存储器。

SRAM 的结构示意图和操作时序图如图 2-21 所示, 通常 SRAM 有 4 种引脚:

- CE 是芯片启用输入信号。CE 在低电平工作。即当  $\overline{CE}=1$  时, SRAM 的 Data 引脚被禁用;  $\overline{CE}=0$  时, SRAM 的 Data 引脚被启用。
- R/W' 是读写控制信号 ( $\overline{W}$  表示低电平有效), 用于控制当前操作是读 ( $R/W'=1$ ) 还是写 ( $R/W'=0$ )。读写通常是相对于 CPU 而言的, 所以读意味着从 RAM 中读出, 写意味着写入到 RAM 当中。有些 SRAM 的读写信号是分开的, 分为两个控制引脚 RD 和 WR。
- Address 是一组地址线, 用于给出读或写的地址。
- Data 是用于数据传输一组双向信号线。当  $R/W'=1$  时, 该引脚为输出; 当  $R/W'=0$  时, 该引脚为输入。



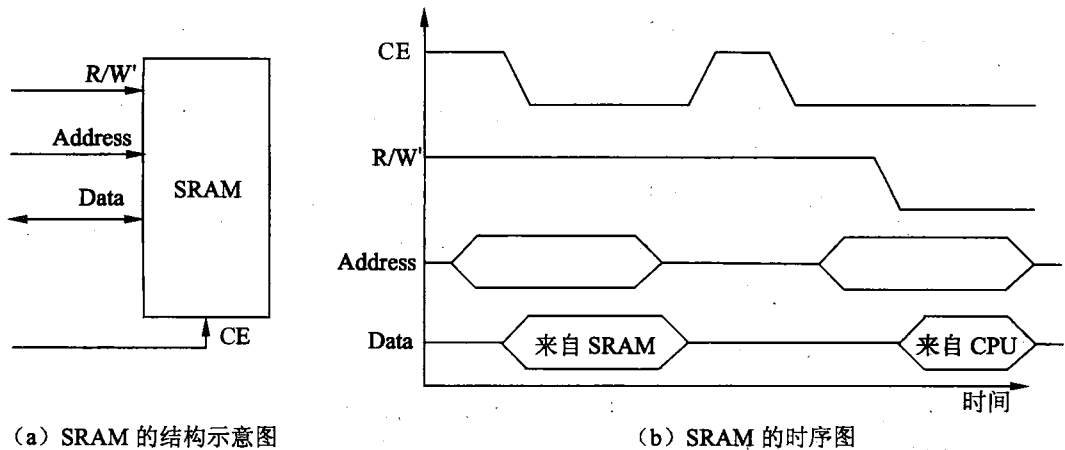


图 2-21 SRAM

SRAM 上的读操作周期如下:

- ① 当  $R/W'=1$  时, 让  $CE=0$  启用该芯片。
- ② 将地址送到地址线上。
- ③ 经过一段时间的延迟之后, 数据出现在数据线上。

SRAM 上的写操作周期如下:

- ① 让  $CE=0$ , 启用该芯片。
- ② 让  $R/W'=0$ 。
- ③ 地址出现在地址线上, 数据出现在数据线上。

## (2) DRAM

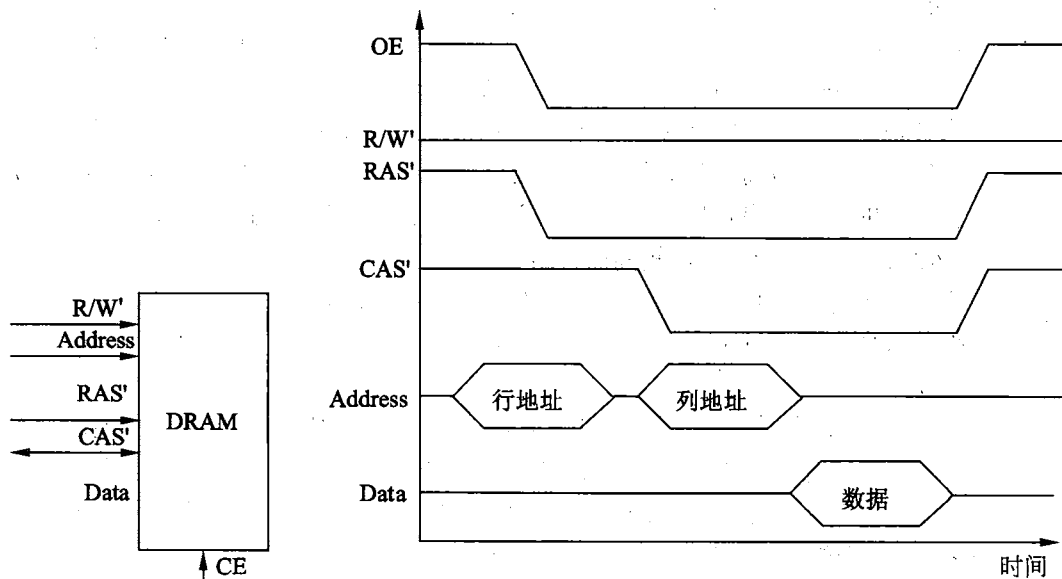
DRAM 表示动态随机存取存储器。这是一种以电荷形式进行存储的半导体存储器。DRAM 中的每个存储单元由一个晶体管和一个电容器组成, 数据存储在电容器中。电容器会由于漏电而导致电荷丢失, 因而 DRAM 器件是不稳定的。为了将数据保存在存储器中, DRAM 器件必须有规律地定时进行刷新。

DRAM 的接口更加复杂, 因为 DRAM 被设计成所需的引脚数最少。基本 DRAM 接口如图 2-22 所示。除了在 SRAM 中的信号线以外, DRAM 还有行地址选择 (RAS) 和列地址选择 (CAS)。需要这些地址信号是因为地址线只提供地址的一半。即当  $RAS'=0$  时, 地址的行部分 (地址的高位部分) 置于地址线; 当  $CAS'=0$  时, 地址的列部分 (地址的低位部分) 置于地址线。DRAM 通常有以下引脚:

- CE 片选信号, 低电平有效。即  $CE=0$ , 选中 DRAM。
- $R/W'$  读写控制信号, 当  $R/W'=1$ , 执行读操作; 当  $R/W'=0$ , 执行写操作。

- RAS'行地址选通信号, 通常接地址的高位部分。
- CAS'列地址选通信号, 通常接地址的地位部分。
- Address 是一组地址线, 用于给出读或写的地址。
- Data 是用于数据传输一组双向信号线。

注意, DRAM 必须被刷新。这是因为它用内部电路系统来存储数据。与 SRAM 不同, 因为芯片上有寄生电阻, 存储在电容上的电荷会泄露。DRAM 上数据的生命期一般为 1ms。可通过执行一次内部读操作来刷新数据。在这个过程中, 原来的值被丢弃。DRAM 一次刷新请求可刷新 DRAM 的一整行。它提供了一种快速刷新模式, 即先 CAS 后 RAS (CAS-before-RAS) 刷新。顾名思义, 这种模式通过设置 RAS=0 之前让 CAS=0 来启动。DRAM 为当前要刷新的行设一计数器。在 RAS 前声明 CAS 导致 DRAM 刷新那一行并更新计数器的值。



(a) 基本 DRAM 的结构示意图

(b) 基本 DRAM 时序图

图 2-22 DRAM

DRAM 的外部逻辑 (即存储控制器) 周期性执行先 CAS 后 RAS 刷新, 让整个存储器在刷新间隔内完成一次刷新。DRAM 和 CPU 之间接口必须考虑刷新, 读写请求直到刷新完成才被满足。存储控制器在 DRAM 和 CPU 之间, 在刷新存储器的间隔期间插入等待状态。

由于程序经常访问同一存储区的不同地址，因此早期为提高 DRAM 性能开发的一种方法就是页模式。页模式访问一次仅提供一个行地址而提供许多列地址。当 CAS 被探测到列地址到来时，RAS=0。页模式一般支持读、写。

页模式的一个改进版本是 EDO（扩展的数据输出）。EDO 访问与页模式类似，它们都允许一个行地址后有多个列地址。EDO 的含义是：数据被保持有效直到 CAS 的下降沿，而不是页模式中的上升沿。

改进 DRAM 性能的另一个方法是引入时钟。同步 DRAM 要求对一个时钟沿引用事件；DRAM 内部电路系统可以工作得更快，因为它不必从异步输入中获得定时信息。同步 DRAM 拥有一般 DRAM 的输入和时钟，如图 2-23 所示。输入（RAS，CAS 等）的改变发生在时钟沿，与 DRAM 输出一样。

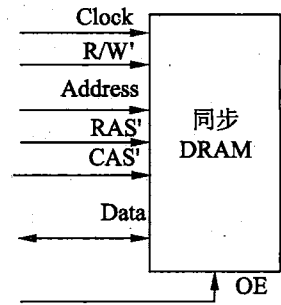


图 2-23 同步 DRAM 的结构示意图

此外，已经为特定的应用开发了带有更复杂接口的其他 RAM 类型：

- 设计视频 RAM，以加速视频操作。它包括标准并行接口和由移位寄存器馈送值的串行接口。移位寄存器将以和并行接口上其他操作并行执行的方式提供对数据一位一位的访问。串行接口一般连到显示器，而并行接口连到微处理器上。
- RAM 总线被设计成性能较高而价格相对较低的 RAM 系统。它包括多个存储区，它们可被并行寻址。许多特性，如控制总线 and 数据总线的分离，有助于使持续数据传输率大大超过 1GB/s。

### (3) DDRAM

随着嵌入式处理器主频的提高，SDRAM 的速度逐渐成了限制系统性能的瓶颈。SDRAM 通常只能工作在 133MHz 主频，而现在很多 32 位处理器的主频已经达到了 200MHz 以上。因此采用新一代内存是嵌入式系统必然的趋势。DDRAM 就是这种需求下出现的，目前已占据了内存技术的主流，且价格便宜，下面对 DDRAM 进行简要介绍。

DDRAM 是基于 SDRAM 技术的，很多指令也是相互兼容的。但 DDRAM 在 SDRAM 的基础上做了较大的改进，在性能上有了很大的提高。现在很多嵌入式处理器已经支持 DDR 内存接口，比如 AMD 公司的 AU1200 处理器，PowerPC 处理器和 Intel 公司用于网络处理的 IXP 系列处理器。

DDRAM 依靠一种叫做双倍预取（2n-prefetch）的技术，即在内存芯片内部的数据宽度是外部接口数据宽度的 2 倍，使峰值的读写速度达到输入时钟速率的 2 倍，并且 DDRAM 允许在时钟脉冲的上升沿和下降沿传输数据，这样不需要提高时钟的频率就能加倍提高

SDRAM 的速度,并具有比 SDRAM 多一倍的传输速率和内存带宽。同时为了保证在高速运行时的信号完整性,DDRAM 技术还采用了差分输入的方式。总的来说,DDRAM 采用了更低的电压、差分输入和双倍数据速率输出等技术。

在 133MHz 下,DDR 内存带宽可以达到  $133 \times 64\text{b} / 8 \times 2 = 2.1\text{GB/s}$ , 200MHz 外频标准出台后,其带宽更是达到了  $200 \times 64\text{b} / 8 \times 2 = 3.2\text{GB/s}$  的海量。

## 2.2.6 外部存储器的种类与选型

### 1. 常见外存的种类

外存储器也称辅助存储器,简称外存或辅存。外存主要指那些容量比主存大、读取速度较慢、通常用来存放需要永久保存的或相对来说暂时不用的各种程序 and 数据的存储器。

在嵌入式系统中常用的外存有:磁盘存储器和光盘存储等。

### 2. 硬盘存储器的基本结构与分类

硬盘存储器具有存储容量大,使用寿命长,存取速度较快的特点。硬盘存储器的硬件包括硬盘控制器(适配器)、硬盘驱动器以及连接电缆。硬盘控制器(Hard Disk Controller,简称 HDC)对硬盘进行管理,并在主机和硬盘之间传送数据。硬盘控制器以适配卡的形式插在主板上或直接集成在主板上,然后通过电缆与硬盘驱动器相连。硬盘驱动器(Hard Disk Drive,简称 HDD)中有盘片、磁头、主轴电机(盘片旋转驱动机构)、磁头定位机构、读/写电路和控制逻辑等。

为了提高单台驱动器的存储容量,在硬盘驱动器内使用了多个盘片,它们被叠装在主轴上,构成一个盘组;每个盘片的两面都可用作记录面,所以一个硬盘的存储容量又称为盘组容量。

#### (1) 硬盘存储器分类

根据头-盘是否是一个密封的整体,硬盘存储器可分为温彻斯特盘和非温彻斯特盘两类。温彻斯特盘是根据温彻斯特技术设计制造的,它的主要特点是磁头、盘片、磁头定位机构、主轴、甚至连读/写驱动电路等都被密封在一个盘盒内,构成一个头-盘组合体。这个组合体不可随意拆卸,它的防尘性能好,可靠性高,对使用环境要求不高。而非温彻斯特盘磁盘的磁头和盘片等不是密封的,因此要求有超静的使用环境,只能用于中型、大型计算机机房中。

根据磁头是否可移动,硬盘存储器可分为固定头硬盘和活动头硬盘两类。固定头硬盘机中,每个磁道对应一个磁头。工作时磁头无径向移动,其特点是存取速度快,省去了磁头寻找磁道的时间,磁头处于加工状态即可开始读/写。但由于磁头太多,使磁盘的道密度不可能很高,而整个磁盘机的造价却比较高。活动头硬盘中,每个盘面上只有一个读/写头,安装在读/写臂上,当需要在不同磁道上读/写时,要驱动读/写臂沿盘面作径向移动。由于

增加了寻道时间, 所以其存取时间比固定头硬盘要长。

## (2) 磁头技术

硬盘读取数据是通过磁头来完成的。最早的传统磁头是电磁感应式磁头, 这些磁头是读写合一的, 由于硬盘读、写操作的不同, 这种二合一磁头就必须同时兼顾到读/写两种特性, 对硬盘的设计造成了不便。后来的硬盘开始采用 MR (磁阻磁头技术) 磁头这种分离式的磁头结构: 写入磁头仍采用磁感应磁头, 而 MR 磁头则作为读取磁头磁阻。这样便可以得到更好的读/写性能。MR 磁头是通过阻值变化来感应信号幅度, 对信号变化相当敏感, 准确性也较高, 而且由于读取的信号幅度与磁道宽度无关, 故磁道可以做得很窄, 从而提高了盘片密度, 扩大了盘片的容量。然而, 随着单碟容量的不断增加, 终于到了 MR 磁头的读取极限, 于是 GMR (巨磁阻磁头技术) 磁头诞生了, 现在单碟容量超过 5GB 的硬盘都采用了 GMR 磁头。进入 2001 年后, 几乎全部硬盘均采用 GMR 磁头。GMR 磁头技术是在 MR 的基础上开发的, 它比 MR 具有更高的灵敏性。正是基于越来越先进的磁头技术, 才使硬盘单碟容量越做越大成为可能, 目前最新的磁头是基于第三代巨磁阻磁头技术。

## (3) 接口

硬盘的接口方式可以说是硬盘另一个非常重要的技术指标, 这点从 SCSI 硬盘和 IDE 硬盘的巨大差价就能体现出来, 接口方式直接决定硬盘的性能。现在最常见的接口有 IDE (ATA) 和 SCSI 两种, 此外还有一些移动硬盘采用了 PCMCIA 或 USB 接口。

- IDE (Integrated Drive Electronics): IDE 接口最初由 CDC、康柏和西部数据公司联合开发, 由美国国家标准协会(ATA)制定标准, 所以又称 ATA 接口。普通用户家里的硬盘几乎全是 IDE 接口的。IDE 接口的硬盘可细分为 ATA-1(IDE)、ATA-2 (EIDE)、ATA-3 (Fast ATA-2)、ATA-4 (包括 UltraATA、Ultra ATA/33、Ultra ATA/66) 与 Serial ATA (包括 Ultra ATA/100 及其他后续的接口类型)。基本 IDE 接口数据传输率为 4.1MB/s, 传输方式有 PIO 和 DMA 两种, 支持总线为 ISA 和 EISA。后来为提高数据传输率、增加接口上能连接的设备数量、突破 528MB 限制及连接光驱的需要, 又陆续开发了 ATA-2、ATAPI 和针对 PCI 总线的 FAST-ATA、FAST-ATA2 等标准, 数据传输率达到了 16.67MB/s。1996 年 Quantum (昆腾) 和 Intel 公司合作开发了 Ultra DMA/33 接口, 严格说来, 这已经不能算 IDE 接口, 而应称为 EIDE 接口, 它采用 PIO 模式, 数据传输率达到 33MB/s。1999 年昆腾又推出了 Ultra DMA/66 接口, 传输率为 Ultra DMA/33 的两倍, 采用 CRC (循环冗余循环校验) 技术以保证数据传输的安全性, 并且使用了 80 线的专用连接电缆, 现在市场上主流的硬盘接口类型即为 Ultra ATA/66。不过, 在进入新世纪后, 最有前景的硬盘接口类型则该是 Ultra ATA/100, 它的理论最大外部数据传输率可以高达 100MB/s。

- SCSI (Small Computer System Interface, 小型计算机系统接口): SCSI 并不是专为硬盘设计的, 实际上它是一种总线型接口。由于独立于系统总线工作, 所以它的最大优势在于其系统占用率极低, 但由于其昂贵的价格, 这种接口的硬盘大多用于服务器等高端应用场合。

### 3. 光盘存储器

相对于利用磁头变化和磁化电流进行读/写的磁盘而言, 用光学方式读/写信息的圆盘称为光盘, 以光盘为存储介质的存储器称为光盘存储器。

#### 1) 光盘存储器的类型

- CD-ROM 光盘: 所谓 CD-ROM (Compact Disc Read Only Memory), 即只读型光盘, 又称固定型光盘。它由生产厂家预先写入数据和程序, 使用时用户只能读出, 不能修改或写入新内容。
- CD-R 光盘: CD-R 光盘采用 WORM (Write One Read Many) 标准, 光盘可由用户写入信息, 写入后可以多次读出; 但只能写入一次, 信息写入后将不能再修改, 所以称为只写一次型光盘。
- CD-RW 光盘: 这种光盘是可以写入、擦除、重写的可逆性记录系统。这种光盘类似于磁盘, 可重复读/写。
- DVD-ROM 光盘: DVD 代表通用数字化多功能光盘 (Digital Versatile Disc), 简称高容量 CD。事实上, 任何 DVD-ROM 光驱都是 CD-ROM 光驱, 即这类光驱既能读取 CD 光盘, 也能读取 DVD 光盘。DVD 除了密度较高以外, 其他技术与 CD-ROM 完全相同。

#### 2) 光盘存储器的组成及工作原理

##### (1) 光盘存储器的组成

光盘存储器由光盘控制器和光盘驱动器及接口组成。光盘控制器主要包括数据输入缓冲器、记录格式器、编码器、读出格式器、数据输出缓冲器等部分。光盘驱动器主要包括主轴电机驱动机构、定位机构、光头装置及电路等。其中光头装置部分最复杂, 是驱动器的关键部分。

光盘片是指整个盘片, 包括光盘的基片和记录介质。基片一般采用碳酸酯晶片制成, 是一种耐热的有机玻璃。无论是只读的 CD-ROM 光盘、DVD-ROM 光盘还是一次可写的 CD-R、可反复擦写的 CD-RW 光盘, 表面上看都是一张 120mm 直径的盘片, 中心有一个用来固定的 15mm 直径小圆孔, 环孔中心半径 13.5mm 范围内和盘片外沿 1mm 内是空白区, 真正存放数据的便是中间一段宽度为 38mm 的环形区域。它们的不同之处主要是这些光盘的记录层 (用于记录数据) 的化学成分存在差异。

## (2) CD-ROM 光盘的制作和读取

CD-ROM 光盘是采用母盘灌制的方法大批量生产的, 首先用事先编制好的程序控制激光刻片机, 对一张玻璃基板进行蚀刻, 将要存储的数据内容在玻璃基板上形成一个个数据凹痕, 这个制作完成的玻璃基板就是大量压制 CD-ROM 光盘的模具。模具制造完成之后, 用聚碳酸酯溶液倒入模具中, 冷却后便变成具有同玻璃基板相应凹槽的基片, 在其表面喷有一层厚度约为 50 nm 的铝质反光涂料, 通常将它称为反射层, 其作用就是将读取数据的激光反射给接收装置; 此外还必须覆盖一层起保护作用的透明基片, 这样盘片的制作就完成了。

CD-ROM 光盘上有一条从内向外的由凹痕和平坦表面相互交替而组成的连续的螺旋形路径。也就是说, 数据和程序都是以刻痕的形式保存在盘片上的。当一束激光照射在盘面上, 靠盘面上有无凹痕的不同反射率来读出程序和数据。一片 CD-ROM 盘上, 存储容量可达到 600MB, 相当于 500 张 1.2MB 的软盘。但是, 因为程序和数据文件是按内螺旋线的规律顺序存放在盘上的, 不能像磁盘驱动器那样读取文件的每个扇区, 所以读取速度较慢。

当光驱读取这些盘片时, 激光头射出的激光束在穿过表面的透明基片后, 直接聚集在盘片射层上, 被反射回来的激光会被光感应器检测到。每当激光通过凹痕时光强会发生变化, 代表读取到数据“1”; 而激光通过平坦表面时光强不发生变化, 则代表读取到数据“0”。光驱的信号接收系统则负责把这种光强的变化转换成相应的电信传送到系统总线, 从而实现数据的读取。

## 4. 标准存储卡 (Compact Flash, CF 卡)

CF 卡是 1994 年由 SanDisk 最先推出的。它在接口上具有 PCMCIA-ATA 功能, 并与之兼容。就是说, CF 卡不但可以工作在 IDE 接口的模式下, 它还可以工作在 PCCard 模式。最初, CF 卡只是利用 Flash 技术的存储卡。接着 CF+卡的标准也被制定出来, 它是 Compact Flash 的衍生技术规格, CF+的物理规格和 CF 完全相同, 但 CF+卡并不是 CF 卡那样的闪存存储器, 而主要是相同规格的 I/O 设备, 如 CF 串口卡、CF Modem、CF 蓝牙、CF USB 卡、CF 网卡、CF GPS 卡、CF GPRS 卡等, 这一类设备在手持设备上用得最多, 使用很方便, 其兼容性也和 CF 卡一样好。但是, 按照 CF+卡标准, 它不一定要支持 ATA 接口。也就是说, 对于 CF+卡, 建议让它工作在 PCMCIA 模式下。CF 卡可以看作是 PCMCIA 卡的一个子集, 可以通过物理上的转换器, 直接转换成 PCMCIA 卡使用。

按照机构尺寸, CF 卡可分为两类: I 型 (Type I) 和 II 型 (Type II), 二者的规格和特性基本相同。只是后者比前者略厚一些 (5.0mm、3.3mm)。CF 卡的 II 型插座, 可以同时兼容这两种类型的卡。

CF 卡可以工作在 PC 卡 ATA I/O 模式、PC 卡 ATA 存储模式和实 IDE 模式 3 种模式下, 实 IDE 模式与 IDE 接口完全兼容。CF 卡遵循 ATA 协议, 属于块存储设备, 存储单元是通过磁头 (head)、柱面 (cylinder, 也称磁道) 和扇区 (sector) 组织起来的; 在物理寻址 (CHS) 方式下, 每一组 H/C/S 参数唯一确定存储卡中的一个扇区, 通常一个扇区拥有 512B 的数据空间。一个驱动器格式化后的容量为磁头数 $\times$ 柱面数 $\times$ 扇区数 $\times$ 512 字节。在物理寻址模式下, 扇区 (S) 是最低的地址单位, 其次是磁头 (H), 最后的柱面 (C) 为最高寻址单位。此外, 还有逻辑寻址方式 (LBA)。在这种寻址方式下, CF 卡按照以连续序列的逻辑扇区编号进行寻址, 主机不必知道 CF 卡的物理几何结构。使用 28 个数据位来表示逻辑扇区的地址, 可以寻址 228 个扇区, 理论上可以寻址 136GB 的容量。下面给出物理寻址方式与逻辑寻址方式的对应关系。设 NS 为每磁道扇区数, NH 为磁头数, C、H、S 分别表示磁盘的柱面、磁头和扇区编号, LBA 表示逻辑扇区号, div 为整除计算, mod 为求余计算, 则:

$$LBA = NH \times NS \times C + NC \times H + S - 1;$$

$$C = (LBA \div NS) \div NH;$$

$$H = (LBA \div NS) \bmod NH;$$

$$S = (LBA \bmod NS) + 1。$$

CF 卡为 50 引脚接口。其中重要的信号线 16 根数据线、11 根地址线 (在 TrueIDE 模式下仅用 3 根地址线)、2 根寄存器组选择信号线 (CS0、CS1)、数据的读写线 (IORD、IOWR)、1 根中断信号请求线 (INTRQ) 和 1 根复位线 (RESET)。

### 5. 安全数据卡 (Secure Digital Card, SD 卡)

SD 卡是由日本 Panasonic 公司、TOSHIBA 公司和美国 SanDisk 公司共同开发研制的全新的存储卡产品。SD 存储卡是一个完全开放的标准 (系统), 多用于 MP3、数码摄像机、数码相机、电子书及 AV 器材等, 尤其是被广泛应用在超薄数码相机上。SD 卡在外形上同 MultiMedia 卡保持一致, 大小尺寸比 MMC 卡略厚, 容量也大很多。并且兼容 MMC 卡接口规范。SD 卡最大的特点就是通过加密功能, 可以保证数据资料的安全保密。它还具备版权保护技术, 所采用的版权保护技术是 DVD 中使用的 CPRM 技术 (可刻录介质内容保护)。

#### (1) SD 存储卡概念

SD 卡通信基于 9 芯的接口 (Clock, Command, 4xDat, 3xPower lines), 最大的操作频率是 25MHz。

#### (2) SD 卡的总线拓扑

SD 卡系统支持两种通信协议: SD 和 SPI 方式。模式的选择对主机是透明的, 由 SD 卡自动检测复位命令的模式, 在此后的通信过程中始终使用此种通信方式。SD 卡在结构上使用一主多从星型拓扑结构。拓扑图如图 2-24 所示。



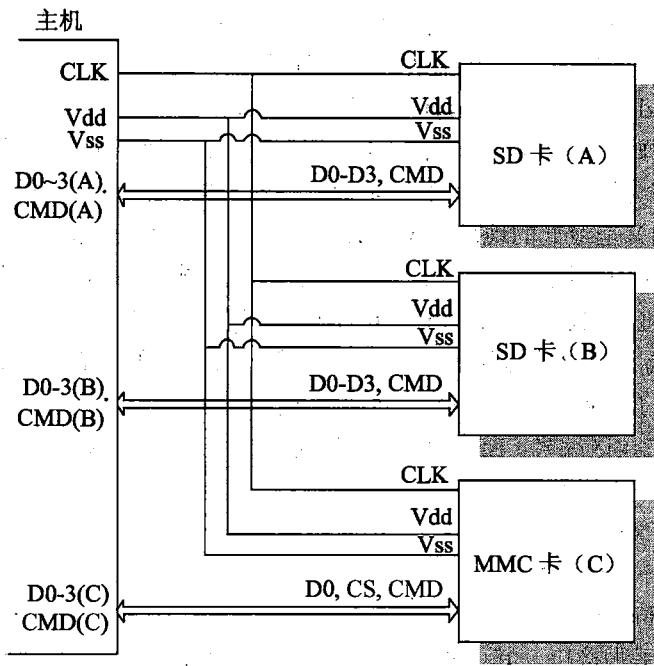


图 2-24 SD 卡系统的总线拓扑图

(3) SD 总线信号

- CLK: 时钟信号。
- CMD: 命令/相应信号。
- DAT0-DAT3: 双向数据传输信号。
- VDD, VSS1, VSS2: 电源和地信号。

其原理图如图 2-25 所示。

(4) SD 卡外形和接口

标准 SD 的外形尺寸是 24mm×32mm×2.1mm，如图 2-26 所示。

SD 卡引脚定义如表 2-9 所示。

表 2-9 SD 卡引脚定义

引脚	SD 模式			SPI 模式		
	名 称	类 型	描 述	名 称	类 型	描 述
1	CD/DAT3	I/O/PP	卡检测/数据线[Bit 3]	CS	I	片选信号
2	CMD	PP	命令/相应	DI	I	数据输入
3	V <sub>SS1</sub>	S	接地	V <sub>SS</sub>	S	接地

续表

引脚	SD 模式			SPI 模式		
	名称	类型	描述	名称	类型	描述
4	VDD	S	供给电压	VDD	S	供给电压
5	CLK	I	时钟	SCLK	I	时钟
6	Vss <sub>2</sub>	S	接地	Vss2	S	接地
7	DAT0	I/O/PP	数据线[Bit 0]	DO	O/PP	数据输出
8	DAT1	I/O/PP	数据线[Bit 1]	RSV		
9	DAT2	I/O/PP	数据线[Bit 2]	RSV		

S: 供电; I: 输入; O: 输出; 使用推挽驱动; PP: I/O 使用推挽方式。

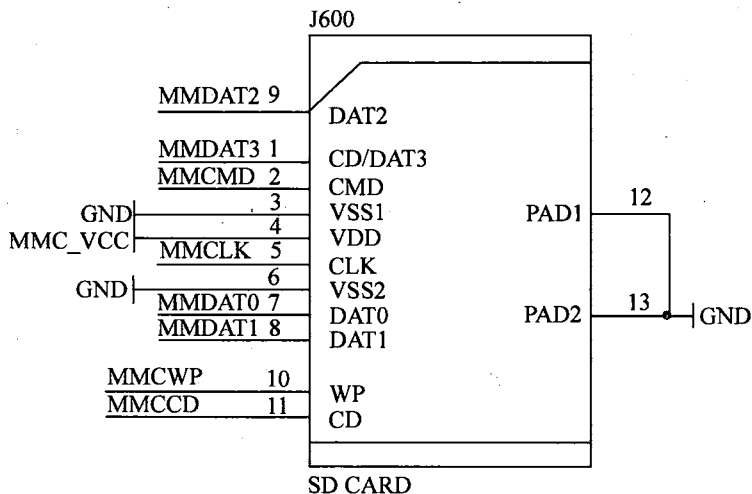


图 2-25 SD 卡原理图

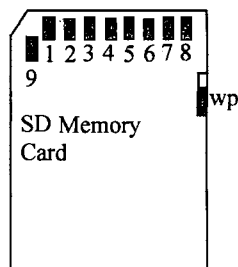
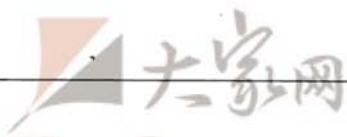


图 2-26 SD 卡的外形和接口

## 2.3 嵌入式系统输入输出设备

### 2.3.1 嵌入式系统常用输入/输出设备概述

在嵌入式系统中，输入/输出设备非常重要。设计上常常需要额外加上扩充内存，以执行更多的指令或是处理更多的数据；或是信号输入的外围设备，如触摸屏、键盘、鼠标、开关或是传感器等；以及信号输出的外围设备，如显示器、打印机或扬声器等。依靠输入/输出接口硬件体系结构，用来连接微处理器与外围设备，以便传递需要的信号输入与输出。



### 1. 键盘、鼠标

嵌入式系统中的键盘往往简化为少数的几个键，鼠标在本质上没有什么变化，键盘负责输入字符或数字，鼠标负责显示位置及选择或取点等操作。手写笔是键盘的扩展，可以代替键盘，方便地进行汉字或字符的输入，但需要相应识别软件的支持。

### 2. 触摸屏

在便携式的电子类嵌入式产品中，触摸屏由于其轻便、占用空间少、方便灵活等优点，已经逐渐取代键盘成为嵌入式系统的输入设备。

### 3. 显示器

嵌入式系统中的显示器一般为小屏幕的 LCD（液晶显示器）或数码管，传统的 CRT 显示器由于其体积庞大，不易携带，正逐渐被 LCD 所取代。LCD 满足了嵌入式系统日益提高的要求，它可以显示汉字、字符和图形，同时还具有低压、低功耗、体积小、重量轻和超薄等很多优点。随着嵌入式系统的应用越来越广泛，功能也越来越强大，对系统中的人机界面的要求也越来越高，在应用需求的驱使下，LCD 在嵌入式系统中的应用越来越广泛。

LCD 基本上分为无源阵列彩显 DSTN-LCD（俗称伪彩显）和薄膜晶体管有源阵列彩显 TFT-LCD（俗称真彩显）。

#### • DSTN (Dual-Layer Super Twist Nematic) -LCD

DSTN LCD 不能算是真正的彩色显示器，因为屏幕上的每个像素的亮度和对比度不能独立控制，它只能显示颜色的深度，与传统的 CRT 显示器的颜色相比相距甚远，因而也被叫做伪彩显。通常应用在对显示要求不高的场合。

#### • TFT (Thin Film Transistor) -LCD

TFT-LCD 的每个液晶像素点都是由集成在像素点后面的薄膜晶体管来控制，使每个像素都能保持一定电压，从而可以做到高速度、高亮度、高对比度的显示。TFT-LCD 是目前最好的 LCD 彩色显示设备之一，是现在高端嵌入式设备上的主流显示设备，如数码相机和数码摄像机等。

### 4. 打印机

打印机将人们需要的结果或图像打印在纸上输出，分为针式打印机、喷墨打印机、激光打印机等。绘图仪是一种特殊的打印机，专门用于精确输出大幅的图形。

### 5. 图形图像摄影输入设备

这些是新兴的设备，包含扫描仪、数码相机、数码摄像机等，这些设备将摄取的影像等信息输入嵌入式计算机中，丰富了嵌入式系统在多媒体方面的应用。现在，大多数的影像设备都使用 USB 接口进行交互，使用非常方便。

### 2.3.2 GPIO 原理与结构

GPIO (General Purpose I/O, 通用 I/O) 是 I/O 的最基本形式。它是一组输入引脚或输出引脚, CPU 对它们能够进行存取。有些 GPIO 引脚能加以编程而改变工作方向。GPIO 的另一传统术语称为并行 I/O (parallel I/O)。图 2-27 所示为双向 GPIO 端口的简化功能逻辑图。为简化图形, 仅画出 GPIO 的第 0 位。图中画出两个寄存器: 数据寄存器 PORT 和数据方向寄存器 DDR。

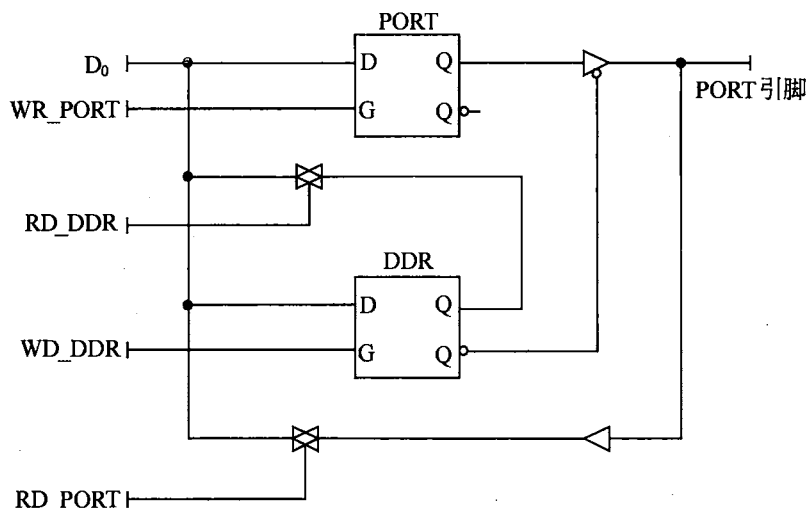


图 2-27 双向 GPIO 功能逻辑图

数据方向寄存器 DDR (Data Direction Register) 设置端口的方向。若该寄存器的输出为 1, 则端口为输出; 若该寄存器的输出为零, 则端口为输入。DDR 状态能够用写入该 DDR 的方法加以改变。DDR 在微控制器地址空间中是一个映射单元。这种情况下, 若要改变 DDR, 则需要将恰当的值置于数据总线的第 0 位即  $D_0$ , 同时激活  $WR\_DDR$  信号。读取 DDR 单元, 就能读得 DDR 的状态, 同时激活  $RD\_DDR$  信号。

若将 PORT 引脚置为输出, 则 PORT 寄存器控制着该引脚状态。若将 PORT 引脚设置为输入, 则此输入引脚的状态由引脚上的逻辑电路层来实现对它的控制。对 PORT 寄存器的写入, 将激活  $WR\_PORT$  信号。PORT 寄存器也映射到微控制器的地址空间。需指出, 即使当端口设置为输入时, 若对 PORT 寄存器进行写入, 并不会对该引脚发生影响。但从 PORT 寄存器的读出, 不管端口是什么方向, 总会影响该引脚的状态。下面为对 PORT 第 0 位进行设置的典型配置:

```
bset  PORTP, BIT0      ;预置 PORTP 数据
bset  DDRP, BIT0       ;预置 PORTP 第 0 位为输出
```

在上面的配置中, 首先配置 PORTP 的第 0 位。这看起来好像顺序颠倒了, 因为尚未将 PP0 配置为输出就先设置 PORTP; 但这样先预置数据寄存器, 可以避免输出端瞬间偶发信号的干扰。然后, 再将 1 写入数据方向寄存器 DDRP 的第 0 位, 用以使 PP0 配置成输出。一旦端口设置成输出, 预置的端口数据就会连接到输出引脚上。

### 2.3.3 A/D 接口基本原理与结构

#### 1. 概述

所谓模/数转换器 (A/D 转换器) 就是把电模拟量转换成为数字量的电路。在当今的现代化生产中, 被广泛应用的实时监测系统和实时控制系统都离不开模/数转换器。图 2-28 给出了微机 and 控制系统接口的框图。一个实时控制系统要实现微机监控实时现场工作过程中发生的各种参数的变化, 首先由传感器把实时现场的各种物理参数 (如温度、流量、压力、PH 值、位移等) 测量出, 并转为相应的电信号, 经过放大、滤波处理, 再通过多路开关的切换和采样/保持电路的保持, 送到 A/D 转换器, 由 A/D 转换器将电模拟信号转换为数字量信号, 之后被微机采集, 微机按一定算法计算输出控制量, 并输出之。输出数据经 D/A 转换器 (数/模转换器) 将数字量转换为电模拟量去控制执行机构。

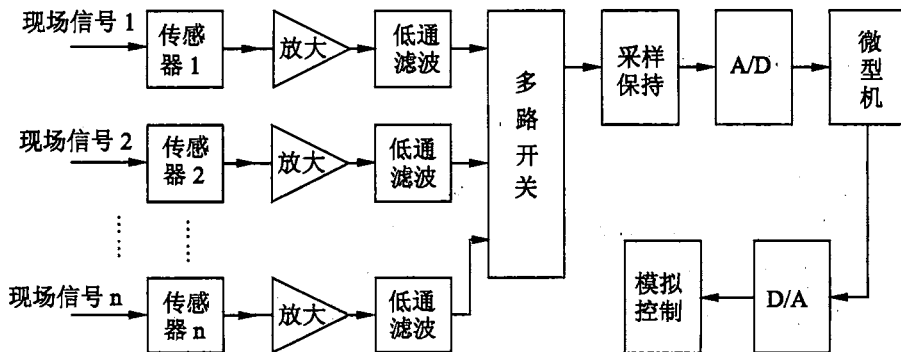


图 2-28 微机与控制系统的接口框图

图 2-28 中各部件的作用归纳如下。

- 传感器：亦称换能器，是把现场各种物理信号按一定规律转换成与其对应的电信号。它是实现测量和控制的首要环节，是测控系统的关键部件。
- 放大器：经传感器转换后的电量如电流、电压的信号幅度很小，很难进行模数转

换, 因此, 必须有放大环节。放大器即把传感器输出的电信号放大到 A/D 转换所需要的量程范围。

- 低通滤波器: 低通滤波器的作用是选出有用的频率信号, 抑制无用的杂散高频干扰, 提高信噪比。
- 多路开关: 多路开关的作用是进行信号切换, 即一次只能把一路信号传送到 A/D 转换器, 当对多路实时现场采集信息时, 多路开关可对多路信号进行切换处理, 控制每次只送一路信息到 A/D 转换器, 实现多路信号的分时处理, 从而降低整个系统的成本。
- 采样/保持电路: 从启动信号转换到转换结束的数字量输出, 经过一定的时间, 而模拟量转换期间, 要求模拟信号保持不变, 所以必用采样保持器。该电路具有两个功能: 采样-跟踪输入信号; 保持-暂停跟踪输入信号, 保持已采集的输入信号, 确保在 A/D 转换期间保持输入信号不变。
- A/D 转换器: 把采样/保持电路锁存的模拟信号转换成数字信号, 等待 CPU 用输入指令读到微机内。

## 2. A/D (模/数) 转换的方法和原理

实现 A/D 转换的方法很多, 常用的方法有计数法、双积分法和逐次逼近法。

### (1) 计数式 A/D 转换法

计数式 A/D 转换的原理如图 2-29 所示。其中,  $V_i$  是模拟输入电压,  $V_o$  是 D/A 转换器的输出电压, C 是控制计数端, 当  $C=1$  时, 计数器开始计数,  $C=0$  时, 则停止计数。  $D_7 \sim D_0$  是数字量输出, 数字输出量又同时驱动一个 D/A 转换器。

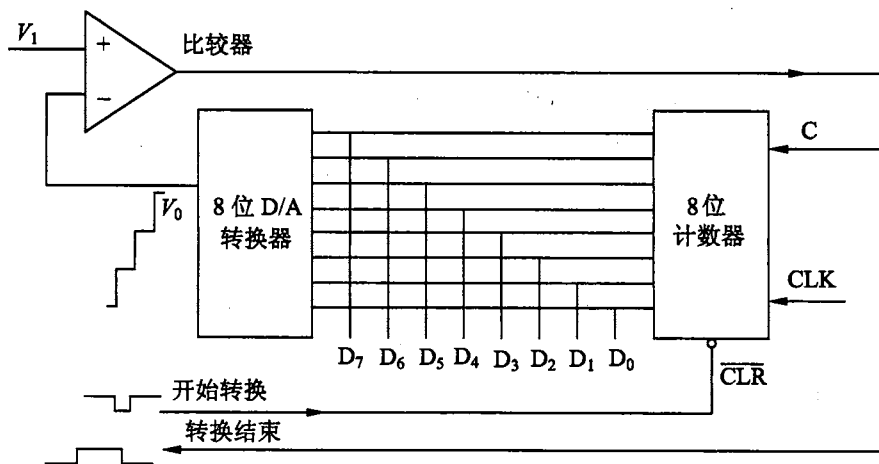


图 2-29 计数式 A/D 转换原理图

具体工作过程如下：首先开始转换信号有效（由高变低），使计数器复位，当开始转换信号恢复高电平时，计数器准备计数。因为计数器已被复位，所以计数器输出数字为 0。这个 0 输出送至 D/A 转换器，使之也输出 0V 模拟信号。此时，在比较器输入端上待转换的模拟输入电压  $V_i$  大于  $V_0$  (0V)，比较器输出高电平，使计数控制信号 C 为 1。这样，计数器开始计数。从此 D/A 转换器输入端得到的数字量不断增加，致使输出电压  $V_0$  不断上升。在  $V_0 < V_i$  时，比较器的输出总是保持高电平。当  $V_0$  上升到某值时，第一次出现  $V_0 > V_i$  的情况，此时，比较器的输出为低电平，使计数控制信号 C 为 0，导致计数器停止计数。这时候数字输出量  $D_7 \sim D_0$  就是与模拟电压等效的数字量。计数控制信号由高变低的负跳变也是 A/D 转换的结束信号，它用来通知计算机，已完成一次 A/D 转换。

计数式 A/D 转换的特点是简单，但速度比较慢，特别是模拟电压较高时，转换速度更慢。当  $C=1$  时，每输入一个时钟脉冲计数器加 1。对一个 8 位 A/D 转换器，若输入模拟量为最大值，计数器从 0 开始计数到 255 时，才转换完毕，相当于需要 255 个计数脉冲周期。对于一个 12 位 A/D 转换器而言，最长的转换周期达 4095 个计数脉冲周期。

## (2) 双积分式 A/D 转换法

双积分式 A/D 转换的基本原理是对输入模拟电压和参考电压进行两次积分，变换成与输入电压均值成正比的时间间隔，利用时钟脉冲和计数器测出其时间间隔，因此，此类 D/A 转换器具有很强的抗工频干扰能力，转换精度高，但速度较慢，通常每秒转换频率小于 10 Hz，主要用于数字式测试仪表，温度测量等方面。双积分式 A/D 转换的电路原理如图 2-30 (a) 所示。电路中的主要部件包括积分器、比较器、计数器和标准电压源。

首先电路对输入待测的模拟电压  $V_i$  进行固定时间的积分，然后换至标准电压进行固定斜率的反向积分，如图 2-30 (b) 所示。反向积分进行到一定时间，便返回起始值。从图 2-30 (b) 中可看出对标准电压进行反向积分的时间  $T$  正比于输入模拟电压，输入模拟电压越大，反向积分回到起始值的时间越长。因此，只要用标准的高频时钟脉冲测定反向积分花费的时间，就可以得到相应于输入模拟电压的数字量，即实现了 A/D 转换。

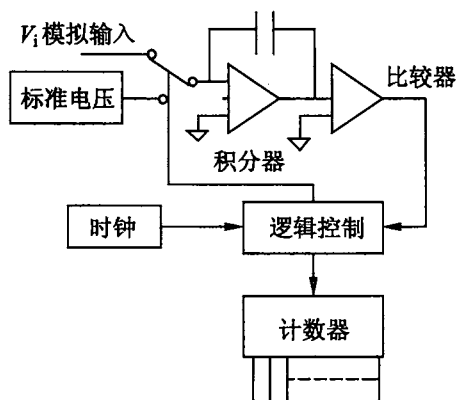
## (3) 逐次逼近式 A/D 转换法

逐次逼近式 A/D 转换法是 A/D 芯片采用最多的一种 A/D 转换方法，和计数式 A/D 转换一样，逐次逼近式 A/D 转换时，是由 D/A 转换器从高位到低位逐位增加转换位数，产生不同的输出电压，把输入电压与输出电压进行比较而实现。不同之处是用逐次逼近式进行转换时，要用一个逐次逼近寄存器存放转换出来的数字量，转换结束时，将最终的数字量送到缓冲寄存器中，其逻辑电路如图 2-31 所示。

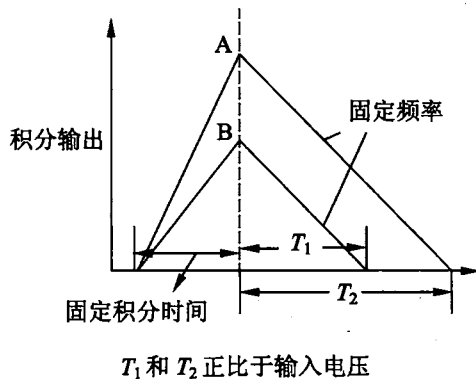
输出为 4 位的逐次逼近 A/D 转换过程如图 2-32 所示。

当  $t_0$  时刻启动信号由高电平变为低电平时，复位逐次逼近寄存器，使之清 0，此时，D/A 转换器输出电压  $V_0$  也为 0，当启动信号由低变为高电平时，转换开始，同时，逐次逼

近寄存器进行计数。



(a) 电路工作原理图



(b) 双积分图式

图 2-30 双积分式 A/D 转换

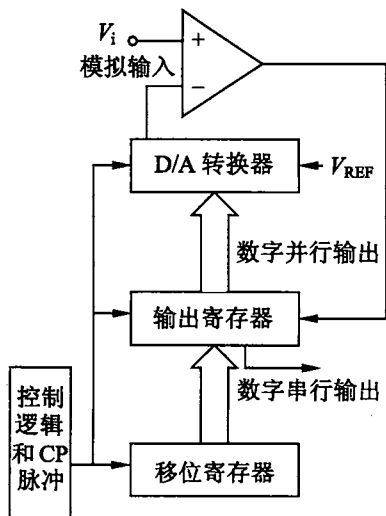


图 2-31 逐次逼近式 A/D 转换原理框图

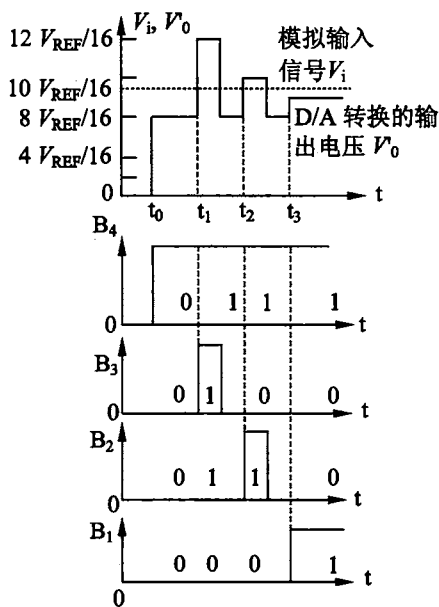


图 2-32 逐次逼近式 A/D 转换过程

逐次逼近寄存器计数时和普通计数器不同，它不是从最低位向高位每次加 1 计数和进位，而是通过类似对分搜索的方式控制逐次比较寄存器进行计数。具体地讲，在启动信号





后第一个时钟脉冲时, 控制电路使逐次逼近寄存器的最高位为 1, 使它的输出为 1000B, 这个数字进入 D/A 转换器, 则其输出电压  $V_0$  为满量程的 128/255。这时, 如果  $V_0$  大于  $V_i$ , 那么, 比较器输出低电平, 控制电路根据此信号清除逐次逼近寄存器中的最高位; 如果  $V_0$  小于  $V_i$ , 比较器输出高电平, 控制电路据此保留最高位的 1, 比较结果是 C 的状态为 1, 则输出寄存器的状态  $B_4$  为 1。此时逐次逼近寄存器的内容为 1000B, 下一个时钟脉冲  $t_1$  时刻控制电路使次高位  $B_3$  为 1。于是, 逐次逼近寄存器的内容为 1100B, 这个数字进入 D/A 转换器输出电压值为满量程的 192 / 255, 此数值与输入电压  $V_i$  比较, 结果  $V_0$  大于  $V_i$ , 则 C 输出状态为 0, 控制电路据此使  $B_3$  位复位, 输出寄存器的状态  $B_3$  为 0, 再下一个时钟脉冲  $t_2$  时刻时, 控制电路使  $B_2$  位为 1, 重复上述过程, 直到  $B_0$  位。经过 4 次比较以后, 逐次逼近寄存器中的数据  $B_4B_3B_2B_1=1001$  就是 A/D 转换后, 与被测(输入)模拟量相应的数字量。

转换结束后, 控制电路送出一个低电平信号作为结束信号, 同时, 将逐次逼近寄存器中的数字量送入缓冲寄存器, 予以输出数字量。

从上面的过程可以分析出, 用逐次逼近法时, 首先使最高位置 1, 这相当于取出最大允许电压的 1/2 与输入电压比较, 如果搜索值在最大允许电压的 1/2 范围内, 那么, 最高位置 0, 否则最高位置 1。之后, 次高位置 1, 相当于在 1/2 的范围中再作对分搜索。如果搜索值超过最大允许电压的 1/2 范围, 那么, 最高位为 1, 次高位也为 1, 这相当于在另外的一个 1/2 范围中再作对分搜索。因此逐次逼近法的计数实质就是对分搜索法。

逐次逼近式 A/D 转换法的特点是速度快, 转换精度较高, 对  $N$  位 A/D 转换只需  $N$  个时钟脉冲即可完成, 一般可用于测量几十到几百微秒的过渡过程的变化, 是计算机 A/D 转换接口中应用最普遍的转换方法。

### 3. A/D 转换的重要指标

#### (1) 分辨率 (Resolution)

分辨率反映 A/D 转换器对输入微小变化响应的能力, 通常用数字输出最低位(LSB)所对应的模拟输入的电平值表示。 $n$  位 A/D 转换能反应  $1/2^n$  满量程的模拟输入电平。由于分辨率直接与转换器的位数有关, 所以一般也可简单地用数字量的位数来表示分辨率, 即  $n$  位二进制数, 最低位所具有的权值, 就是它的分辨率。

值得注意的是, 分辨率与精度是两个不同的概念, 不要把两者相混淆。即使分辨率很高, 也可能由于温度漂移、线性度等原因, 而使其精度不够高。

#### (2) 精度 (Accuracy)

精度有绝对精度 (Absolute Accuracy) 和相对精度 (Relative Accuracy) 两种表示方法。

##### ① 绝对精度:

在一个转换器中,对应于一个数字量的实际模拟输入电压和理想的模拟输入电压之差并非是一个常数。把它们之间的差的最大值,定义为“绝对误差”。通常以数字量的最小有效位(LSB)的分数值来表示绝对精度,如 $\pm 1\text{LSB}$ 。绝对误差包括量化精度和其他所有精度。

### ② 相对精度

是指整个转换范围内,任一数字量所对应的模拟输入量的实际值与理论值之差,用模拟电压满量程的百分比表示。

例如,满量程为10V,10位A/D芯片,若其绝对精度为 $\pm 1/2\text{LSB}$ ,则其最小有效位的量化单位:9.77mV,其绝对精度为 $\pm 4.88\text{mV}$ ,其相对精度为0.048%。

### ③ 转换时间(Conversion Time)

转换时间是指完成一次A/D转换所需的时间,即由发出启动转换命令信号到转换结束信号开始有效的时间间隔。

转换时间的倒数称为转换速率。例如AD570的转换时间为25 $\mu\text{s}$ ,其转换速率为40kHz。

### ④ 量程

量程是指所能转换的模拟输入电压范围,分单极性、双极性两种类型。

例如,单极性的量程为0~+5V,0~+10V,0~+20V;双极性的量程为-5~+5V,-10~+10V。

## 2.3.4 D/A 接口基本原理与结构

### 1. D/A(数/模)转换器的工作原理

D/A转换器的主要功能是将数字量转换为模拟量。数字量是由若干数位构成的,每个数位都有一定的权,如8位二进制数的最高位 $D_7$ 的权为 $2^7=128$ ,只要 $D_7=1$ 就表示具有了128这个值。把一个数字量变为模拟量,就是把每一位上的代码按照权转换为对应的模拟量,再把各位所对应的模拟量相加,所得到各位模拟量的和便是数字量所对应的模拟量。

基于上述思路,在集成电路中,通常采用T型网络实现将数字量转换为模拟电流,然后再用运算放大器完成模拟电流到模拟电压的转换。所以,要把一个数字量转换为模拟电压,实际上需要两个环节:即先由D/A转换器把数字量转换为模拟电流,再由运算放大器将模拟电流转换为模拟电压。目前D/A转换集成电路芯片大都包含了这两个环节,对只包含第一个环节的D/A芯片,就要外接运算放大器才能转换为模拟电压。

#### (1) T型电阻解码网络

在实用的集成电路中,经常采用T型电阻解码网络形式。T型电阻解码网络如图2-33所示,从图中很清楚地看到整个网络中只需要R和2R两种阻值的电阻。

从图2-33中可以看到,对每一个开关 $S_i$ ( $i=0, 1, 2, 3$ )来说,其动端不是接地,便是接运算放大器的虚地,可以认为它们的电位相同,都为“地”。因而开关动端的位置不影响参

考电源  $V_{\text{REF}}$  的总电流和各支路的电流,但是,只有动端和右边的结点相接时,才能给运算放大器的输入端提供电流,下面分析各支路的电流。

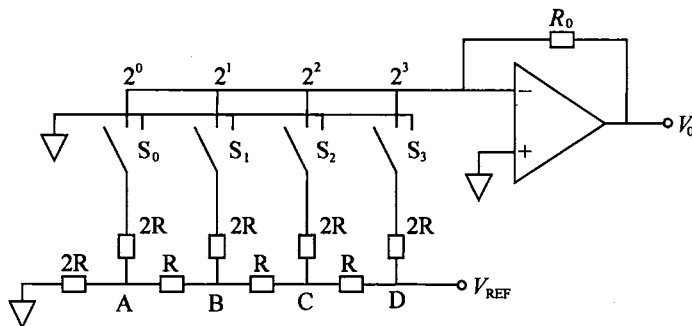


图 2-33 采用 T 型电阻解码网络的 D/A 转换器

T 型电阻解码网络中,节点 A 的左边为两个  $2R$  的电阻并联,它们的等效电阻为  $R$ ,节点 B 的左边也是两个  $2R$  的电阻并联,等效电阻也是  $R$ ,依次类推,最后的 D 点等效于一个电阻  $R$  连接在标准参考电压  $V_{\text{REF}}$  上。根据分压原理 C 点、B 点、A 点的电位分别为  $-V_{\text{REF}}/2$ 、 $-V_{\text{REF}}/4$ 、 $-V_{\text{REF}}/8$ 。已知各点的电位,根据分流原理,知右边第一个支路,即  $S_3$  定端支路的电流为  $-V_{\text{REF}}/2R$ ,右边第二个支路,即  $S_2$  定端支路的电流为  $-V_{\text{REF}}/4R$ ,同理,  $S_1$  和  $S_0$  定端支路的电流分别为  $-V_{\text{REF}}/8R$  和  $-V_{\text{REF}}/16R$ 。

设  $S_0$ 、 $S_1$ 、 $S_2$ 、 $S_3$  分别为各位数码的变量,且  $S_i=1$  表示开关动端接通右结点,  $S_i=0$  则表示开关动端接通左结点,故知运算放大器输入电流为:

$$I = -V_{\text{REF}}/2R \cdot S_3 - V_{\text{REF}}/4R \cdot S_2 - V_{\text{REF}}/8R \cdot S_1 - V_{\text{REF}}/16R \cdot S_0$$

$$I = -V_{\text{REF}}/2R \cdot (2^{-0} \cdot S_3 + 2^{-1} \cdot S_2 + 2^{-2} \cdot S_1 + 2^{-3} \cdot S_0)$$

$$I = -V_{\text{REF}}/2^4 R \cdot (2^3 \cdot S_3 + 2^2 \cdot S_2 + 2^1 \cdot S_1 + 2^0 \cdot S_0)$$

将数码推广到有  $n$  位的情况,可得输出模拟量与输入数字量之间关系的一般表达式:

$$I = -V_{\text{REF}}/2^n R \cdot (S_{n-1} \cdot 2^{n-1} + S_{n-2} \cdot 2^{n-2} + \dots + S_1 \cdot 2^1 + S_0 \cdot 2^0)$$

运算放大器的相应输出电压为:

$$V_O = IR_O = -V_{\text{REF}}R_O/2^n R (S_{n-1}2^{n-1} + S_{n-2}2^{n-2} + \dots + S_12^1 + S_02^0)$$

上式表明,输入数字量被转换成模拟电压  $V_O$ ,输出电压  $V_O$  除了和输入的二进制数有关外,还和运算放大器的反馈电阻  $R_O$  和标准参考电压  $V_{\text{REF}}$  有关,它们之间存在一定的比例关系,其比例系数为  $V_{\text{REF}}R_O / 2^n R$ 。通常,电阻  $R$  在设计 D/A 时已经确定,所以一般不可改变,而在应用时,取不同的标准参考电压和反馈电阻  $R_O$  可以调节输出电压的范围和满刻度量程等。

## (2) 数/模转换器的分类

### ① 电压输出型 (如 TLC5620)

电压输出型 D/A 转换器虽有直接从电阻阵列输出电压的, 但一般采用内置输出放大器以低阻抗输出。直接输出电压的器件仅用于高阻抗负载, 由于无输出放大器部分的延迟, 故常作为高速 D/A 转换器使用。

### ② 电流输出型 (如 THS5661A)

电流输出型 D/A 转换器很少直接利用电流输出, 大多外接电流-电压转换电路得到电压输出, 后者有两种方法: 一是只在输出引脚上接负载电阻而进行电流-电压转换, 二是外接运算放大器。用负载电阻进行电流-电压转换的方法, 虽可在电流输出引脚上出现电压, 但必须在规定的输出电压范围内使用, 而且由于输出阻抗高, 所以一般外接运算放大器使用。此外, 大部分 CMOS D/A 转换器当输出电压不为零时不能正确动作, 所以必须外接运算放大器。当外接运算放大器进行电流电压转换时, 则电路构成基本上与内置放大器的电压输出型相同, 这时由于在 D/A 转换器的电流建立时间上加入了运算放大器的延迟, 使响应变慢。此外, 这种电路中运算放大器因输出引脚的内部电容而容易起振, 有时必须作相位补偿。

### ③ 乘算型 (如 AD7533)

D/A 转换器中有使用恒定基准电压的, 也有在基准电压输入上加交流信号的, 后者由于能得到数字输入和基准电压输入相乘的结果而输出, 因而称为乘算型 D/A 转换器。乘算型 D/A 转换器一般不仅可以进行乘法运算, 而且可以作为使输入信号数字化地衰减的衰减器及对输入信号进行调制的调制器使用。

## 2. D/A 转换器的主要指标

(1) 分辨率 (Resolution): 指最小模拟输出量 (对应数字量仅最低位为 “1”) 与最大量 (对应数字量所有有效位为 “1”) 之比。

(2) 建立时间 (Setting Time): 是将一个数字量转换为稳定模拟信号所需的时间, 也可以认为是转换时间。D/A 转换中常用建立时间来描述其速度, 而不是 A/D 转换中常用的转换速率。一般地, 电流输出 D/A 转换建立时间较短, 电压输出 D/A 转换则较长。

其他指标还有线性度 (Linearity)、转换精度、温度系数/漂移。

## 2.3.5 键盘接口基本原理与结构

### 1. 键盘的分类

键盘的结构通常有两种形式: 线性键盘和矩阵键盘。在不同的场合下, 这两种键盘均得到了广泛的应用。

线性键盘由若干个独立的按键组成, 每个按键的一端与微机的一个 I/O 口相连。有多

少个键就要有多少根连线与微机的 I/O 口相连, 因此, 只适用于按键少的场合。

矩阵键盘的按键按  $N$  行  $M$  列排列, 每个按键占据行列的一个交点, 需要的 I/O 口数目是  $N+M$ , 容许的最大按键数是  $N \times M$ 。显然, 矩阵键盘可以减少与微机接口的连线数, 简化结构, 是一般微机常用的键盘结构。根据矩阵键盘的识键和译键方法的不同, 矩阵键盘又可以分为非编码键盘和编码键盘两种。

非编码键盘主要用软件的方法识键和译键。根据扫描方法的不同, 可以分为行扫描法、列扫描法和反转法 3 种。

编码键盘主要用硬件来实现键的扫描和识别, 通常使用 8279 专用接口芯片, 在硬件上要求较高。

键盘的按键实际上就是开关, 制造这种键的方法是多种多样的, 以下对几种常用的按键开关构造和操作进行介绍。

#### (1) 机械式按键

这类按键开关的构造有两种。一种是内含两个金属片和一个复位弹簧, 按键时, 两个金属片便被压在一起; 另一种机械式按键是用底面带一小块导电橡胶的成型泡沫硅橡胶帽做的, 压键时, 导电橡胶将印制电路板上的两条印制线短路。

机械式按键的缺点是容易产生抖动, 即在触点可靠地接触之前会通断多次; 另外, 触点也会随着时间的推移而变脏或氧化, 使导通的可靠性降低。但该类按键价格较低, 手感好, 使用范围较广。

#### (2) 电容式按键

电容式按键由印制电路板上的两小块金属片和 In 泡沫橡胶片下面可活动的另一块金属片构成。压键时, 可活动的金属片向两块固定的金属片靠近, 从而改变了两块固定的金属片之间的电容。此时, 检测电容变化的电路就会产生一个逻辑电平信号, 以表示该键已被按下。显然, 该类按键没有机械触点被氧化或变脏的问题。

#### (3) 薄膜式按键

这是一种特殊的机械式按键开关, 由三层塑料或橡胶夹层结构构成。上层在每一行键下面有一条印制银导线, 中间层在每个键下面有一个小圆孔, 下层在每一列键下面也有一条印制银导线。压键时将上面一层的印制银导线压过中层的小孔与下面一层的印制银导线接触。此类按键可以做成很薄的密封单元, 更适合于环境较脏的应用中。

#### (4) 霍耳效应按键

霍耳效应按键利用活动电荷在磁场中的偏转效果。参考电流从半导体晶体的两个相对面之间流过, 压键时, 晶体便在磁力线垂直于参考电流方向的磁场中移动。晶体在磁场中移动会在晶体另外两个相对的表面之间产生一个小电压, 该电压经过放大之后用来表示键已被压下。该类按键是一种无机械触点的按键开关, 且密封性很好, 但价格较高。

## 2. 用 ARM 芯片实现键盘接口

为了识别键盘上的闭合键，通常采用两种方法：一种为行扫描法，另一种是行反转法。行扫描法是使键盘上某一行线为低电平，而其余行接高电平，然后读取列值，如果列值中有某位为低电平，则表明行列交点处的键被按下；否则扫描下一行，直到扫描完全部的行线为止。

行反转法识别闭合键时，要将行线接一个并行端口，先让它工作在输出方式下，将列线也接一个并行端口，先让它工作在输入模式方式下。程序使 CPU 通过输出端口往各行线上全部送低电平，然后读入列线的值。如果此时有某一键被按下，则必定会使某一列线值为 0。然后，程序再对两个并行端口进行方式设置，使行线工作在输入方式，列线工作在输出方式，并且将刚才读得的列线值从列线所接的并行端口输出，再读取行线上的输入值，那么，在闭合键所在的行线上的值必为 0。这样，当一个键被按下时，必定可以读取一对唯一的行值和列值。

本应用实例中，要与 4×4 的矩阵键盘接口，采用了节省口线的“行扫描法”方法来检测键盘，这样只需要 8 根口线，在此选取 PF 口作为检测键盘用端口，并设置 PF0~PF3 为输出扫描码的端口，PF4~PF7 为键值读入口。

如图 2-34 所示，16 个按键与 ARM 芯片连接接口逻辑电路图。

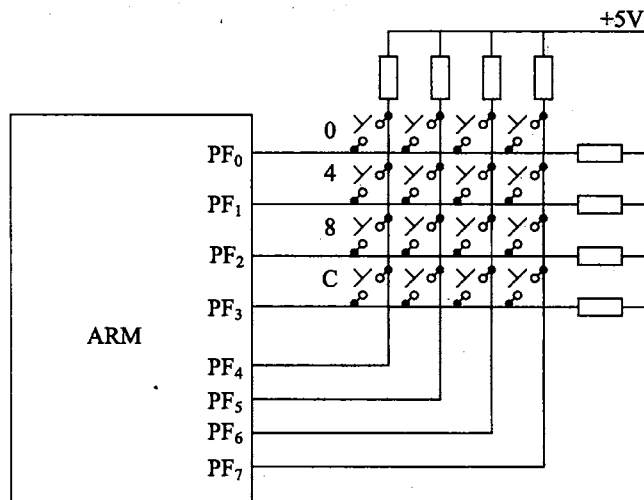


图 2-34 ARM 微处理器实现的键盘接口电路

如图 2-34 所示，按键设置在行、列交叉点上，行、列分别连接到按键开关的两端。列线通过上拉电阻接到 +5V 上。平时无按键动作时，列线处于高电平状态；而当有键按下时，





列线电平状态将由通过此按键的行线电平决定：行线电平如果为低，列线电平为低；行线电平如果为高，则列线电平亦为高。这一点是识别矩阵式键盘是否被按下的关键所在。因各按键之间相互发生影响，所以必须将行、列线信号配合起来并作适当的处理，才能确定闭合键的位置。

根据行扫描法的原理，很容易得出矩阵键盘按键的识别方法，此方法分两步进行：

(1) 识别键盘哪一行的键被按下：让所有行线均为低电平，检查各列线电平是否为低，如果有列线为低，则说明该列有键被按下，否则说明无键被按下。

(2) 如果某列有键被按下，识别键盘哪一行的键被按下：逐行置低电平，并置其余各行为高电平，检查各列线电平的变化，如果列电平变为低电平，则可确定此行此列交叉点处按键被按下。

## 2.3.6 显示接口基本原理与结构

### 1. LCD 显示接口原理与结构

液晶显示器 (Liquid Crystal Display, LCD) 具有耗电省、体积小等特点，被广泛应用于嵌入式系统中。液晶得名于其物理特性：它的分子晶体，以液态而非固态存在。这些晶体分子的液体特性使得它具有两种非常有用的特点：

- 如果让电流通过液晶层，这些分子将会以电流的流向方向进行排列，如果没有电流，它们将会彼此平行排列。
- 如果提供了带有细小沟槽的外层，将液晶倒入后，液晶分子会顺着槽排列，并且内层与外层以同样的方式进行排列。

液晶的第三个特性是很神奇的：液晶层能使光线发生扭转。液晶层表现得有些类似偏光器，这就意味着它能够过滤除了那些从特殊方向射入之外的所有光线。此外，如果液晶层发生了扭转，光线将会随之扭转，以不同的方向从另外一个面中射出。

液晶的这些特点使它可以被用来当作一种开关——既可以阻碍光线，也可以允许光线通过。液晶单元的底层是由细小的脊构成的，这些脊的作用是让分子呈平行排列。上表面也是如此，在这两侧之间的分子平行排列，不过当上下两个表面之间呈一定的角度时，液晶随着两个不同方向的表面进行排列，就会发生扭曲。结果便是这个扭曲的螺旋层使通过的光线也发生扭曲。如果电流通过液晶，所有的分子将会按照电流的方向进行排列，这样就会消除光线的扭转。如图 2-35 所示，如果将一个偏振滤光器放置在液晶层的上表面，扭转的光线通过（如 (a)），而没有发生扭转的光线（如 (b)）将被阻碍。因此可以通过电流的通断改变 LCD 中的液晶排列，使光线在加电时射出，而不加电时被阻断。也有某些设计为了省电的需要，有电流时，光线不能通过，没有电流时，光线通过。

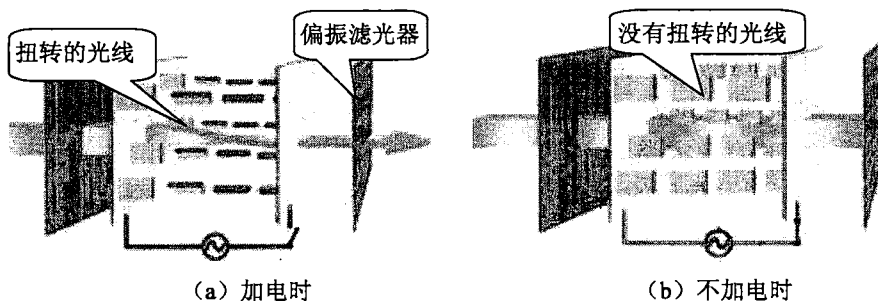


图 2-35 光线穿过与阻断示意图

LCD 显示器的基本原理就是通过给不同的液晶单元供电，控制其光线的通过与否，从而达到显示的目的。在 LCD 显示器中，显示面板薄膜被分成很多小栅格，每个栅格由一个电极控制，通过改变栅格上的电极就能控制栅格内液晶分子的排列，从而控制光路的导通。彩色显示利用三原色混合的原理显示不同的色彩：彩色 LCD 面板中，每一个像素都是由 3 格液晶单元格构成的，其中每一个单元格前面都分别有红色、绿色或蓝色的过滤片，光线经过过滤片的处理变成红色、蓝色或者绿色，利用三原色的原理组合出不同的色彩。

## 2. 电致发光

LCD 的发光原理是通过控制加电与否来使光线通过或挡住，从而显示图形。光源的提供方式有两种：透射式和反射式。笔记本电脑的 LCD 显示器即为透射式，屏后面有一个光源，因此外界环境可以不需要光源。而一般微控制器上使用的 LCD 为反射式，需要外界提供光源，靠反射光来工作。电致发光 (EL) 是液晶屏提供光源的一种方式。电致发光的特点是低功耗，与二极管发光比较而言体积小。

电致发光是将电能直接转换为光能的一种发光现象。电致发光片是利用此原理经过加工制作而成的一种发光薄片，如图 2-36 所示。其特点是：超薄、高亮度、高效率、低功耗、低热量、可弯曲、抗冲击、长寿命、多种颜色选择等。因此，电致发光片被广泛应用于各个领域。

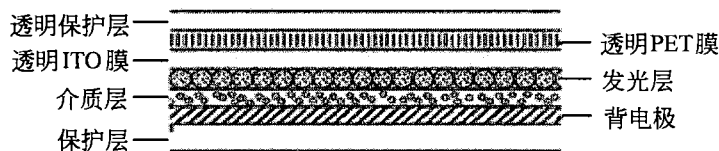


图 2-36 电致发光片的基本结构



### 3. LCD 种类

按照液晶驱动方式分类, 可将目前常见的 LCD 分为扭转向列 (Twist Nematic, 简称 TN) 型、超扭曲向列 (Super Twisted Nematic, 简称 STN) 型和薄膜晶体管 (Thin Film Transistor, 简称 TFT) 3 大类。

TN 型 LCD 是最早商用的 LCD, 也是目前应用最广泛的 LCD 类型, 因为价格便宜, 被广泛应用于手表、时钟、电子计算机、电话、传真机以一般家电用品的数字显示。TN 型 LCD 的基本原理类似于上面所述的 LCD 显示原理, 但它的分辨率很低, 一般用于显示数字、字符等, 很难用于显示图形图像。目前单纯矩阵驱动的 TN 型产品以小尺寸黑白文字显示类 LCD 为主。

STN 型 LCD 是通过改变液晶材料的化学成分, 使液晶分子发生不止一次地扭转, 光线扭转达到  $180^{\circ}\sim 270^{\circ}$ , 从而大大改善了画面的显示品质。为了显示图像, STN 型 LCD 将液晶单元排成阵列, 用输入的信号依次去驱动每一行的电极, 于是当某一行被选定的时候, 列向上的电极被触发打开位于行和列交叉点上的那些像素, 从而控制单个像素的开关。这种显示方式类似于 CRT 的扫描方式, 即同一时刻 STN 型 LCD 上只有一点受控。如果要显示彩色图像, 则把每个像素点分成 RGB 3 个像素点, 并在这 3 个像素点上的光路上增加相关滤光片。STN 型 LCD 的缺点为: 如果有太大的电流通过某个单元, 附近的单元就会受到影响, 引起虚影。如果电流太小, 单元的开和关就会变得迟缓, 降低对比度并丢失移动画面的细节, 而且随着像素单元的增加, 驱动电压也相应提高, 因此 STN 型 LCD 很难做出高分辨率的产品, 它的应用也局限于一些对图像分辨率和色彩要求不是很高的领域。目前以移动电话、PDA、掌上型电脑、汽车导航系统、电子词典等高品质产品中的小尺寸电子显示为主, 不过在这些领域也越来越受到 TFT 型 LCD 的冲击。

TFT 型 LCD 和 STN 型 LCD 相比, 多了一层薄膜晶体管 (TFT) 阵列, 每一个像素都对应一个薄膜晶体管, 控制液晶的电压直接加在这个晶体管上, 再通过晶体管去控制液晶的电压。这个晶体管就好像是一个小“百叶窗”的开关, 而这每个小“百叶窗”又控制着光线的通过率。由于 TFT 型 LCD 的每个像素都可通过点脉冲直接控制, 因此每个节点相对独立, 并可连续控制, 这样晶体管可以迅速地控制每个单元, 每个单元的电干扰很小, 所以可以使用大电流, 而不会有虚影和拖尾现象, 而且更大的电流会提供更好的对比度、更锐利和更明亮的图像, 同时在灰度控制上也非常精确。除此之外, 由于晶体管本身所具有的电容作用, 加载在晶体管上的控制电压不会马上消失, 使得 TFT 型 LCD 对刷新频率的要求大大降低, 也使得它的画面更加稳定, 没有闪烁感。TFT 型 LCD 的结构, 如图 2-37 所示。

在具体的 TFT 技术上, 传统的 TFT 型 LCD 采用半导体制造技术, 在导电玻璃上用非晶硅 (a-Si) 制成薄膜晶体管。由于多晶硅 (Poly-Si) 的电子迁移率约为非晶硅的 100 倍,

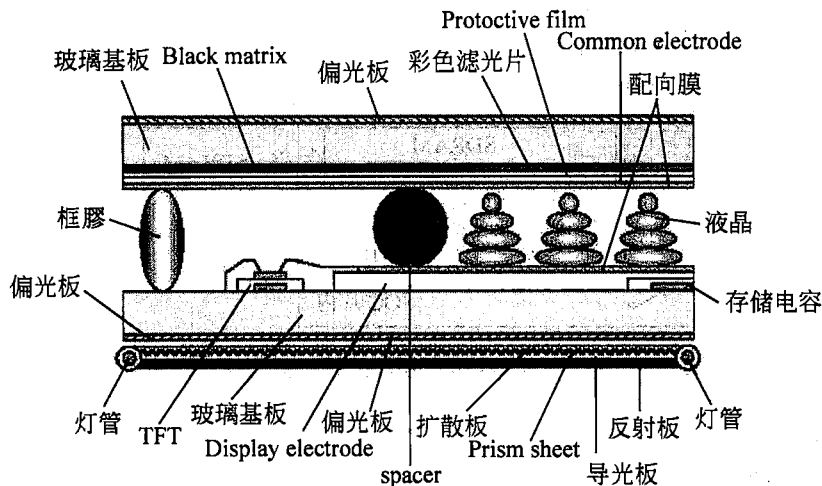


图 2-37 TFT 型 LCD 的结构

面板反应时间比非晶硅快 10 倍，且低温多晶硅（Low Temperature-silicon, LTPS）的开口率较高，画面亮度较高，并可以将周边驱动 IC 与像素一体成型于玻璃基板上，可缩小面板厚度 10%~20%，当技术成熟时，其成本仅有非晶硅的 60% 左右，将成为未来 TFT 型 LCD 的主流产品。但由于量产制造工艺尚不成熟，成品率偏低，制造技术和工艺还有待提高。

TFT 型 LCD 因反应快、显示品质较佳，适用于大型动画显示，被广泛应用于笔记本电脑、计算机显示器、液晶电视、液晶投影机及各式大型电子显示器等产品。近年来随着手机、PDA、数码相机和数码摄像机等手持类设备对显示屏的要求不断提高，TFT 型 LCD 在这些领域也有了越来越广泛的应用。

#### 4. LCD 的设计方法

早期单片机系统，集成度比较低，可扩展接口少，LCD 往往是通过 LCD 控制器连在单片机总线上，或者通过并行接口、串行接口和单片机相连。现在很多厂商都在 SOC 中集成了 LCD 控制器，方便了开发人员有效的使用 LCD。早期低端的芯片提供的一般都是 TN 类型的 LCD 控制器，目前已经有越来越多的芯片提供度 TFT 型显示器的支持。集成了 LCD 控制器的嵌入式处理器（片上系统）的基本体系结构如图 2-38 所示。

处理器内核是整个片上系统的核心，如 ARM 的内核、MIPS 的内核等。系统总线是指处理内部的总线，比如 ARM 著名的 AMBA 总线，其他片上系统的外设都通过总线和处理器连接。LCD 控制器工作的时候，通过 DMA 请求占用系统总线，直接通过 SDRAM 控制器读取 SDRAM 中指定地址（显示缓冲区）的数据。此数据经过 LCD 控制器转换成液晶屏扫描数据的格式，直接驱动液晶屏显示。

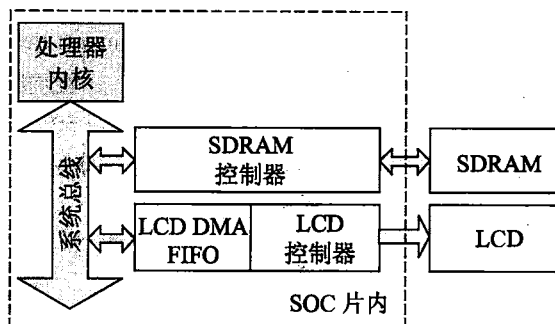


图 2-38 集成了 LCD 控制器的嵌入式处理器体系结构

市面上出售的 LCD 有两种类型：

- 带有驱动电路的 LCD 显示模块，这种 LCD 可以方便地与各种低档单片机进行接口，如 8051 系列单片机，但是由于硬件驱动电路的存在，体积比较大。这种模式常常使用总线方式来驱动。
- LCD 显示器，没有驱动电路，需要与驱动电路配合使用，如图 2-39 所示。特点是体积小，但却需要另外的驱动芯片。也可以使用带有 LCD 驱动能力的高档 MCU 驱动，如 Intel 公司的 XScale 系列的 PXA270 微处理器。

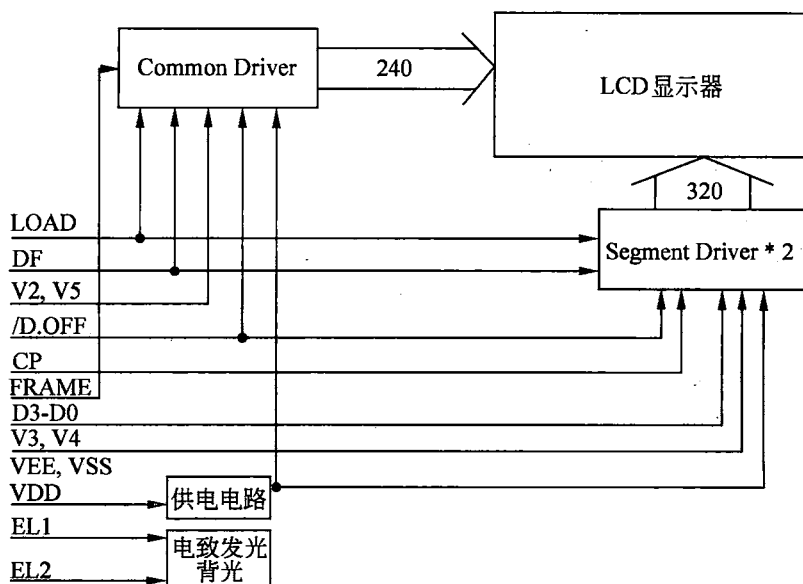


图 2-39 不带驱动电路的 LCD 结构

### (1) 总线驱动方式

一般带有驱动模块的 LCD 显示屏使用这种驱动方式, 由于 LCD 已经带有驱动硬件电路, 因此模块给出的是总线接口, 便于与单片机的总线进行连接。驱动模块具有 8 位数据总线, 外加一些电源接口和控制信号。而且自带显示缓存, 只需要将要显示的内容送到显示缓存中就可以实现内容的显示。由于只有 8 条数据线, 因此常常通过引脚信号来实现地址与数据线复用, 以达到把相应数据送到相应显示缓存的目的。

### (2) 控制器扫描方式

以 PXA270 微处理器为例, PXA270 具有内置的 LCD 控制器, 它具有将显示缓存(在系统存储器中)中的 LCD 图像数据传输到外部 LCD 驱动电路的逻辑功能。支持 DSTN(被动矩阵或叫无源矩阵)和 TFT(主动矩阵或叫有源矩阵)两种 LCD, 并支持黑白和彩显。

在灰度 LCD 上, 使用基于时间的抖动算法(Time-based Dithering Algorithm, 简称 TBDA)和帧率控制(Frame Rate Control, 简称 FRC)方法, 可以支持单色、2 级、4 级和 8 级灰度模式的灰度 LCD。在彩色 LCD 上, 可以支持 16777216 色(24 位)。有 7 路 DMA 通道, 可支持两个 LCD。对于不同尺寸的 LCD, 具有不同数量的垂直和水平像素、数据接口的数据宽度、接口时间及刷新率, 而 LCD 控制器可以进行编程控制相应的寄存器值, 以适应不同的 LCD 显示屏。

PXA270 处理器内置的 LCD 控制器提供了下列外部接口信号。

- L\_FCLK\_RD: LCD 控制器和 LCD 驱动器之间的帧同步信号。它通知 LCD 新的一帧的显示, LCD 控制器在一个完整帧的显示后发出 VFRAME 信号。
- L\_LCLK\_A0: LCD 控制器和 LCD 驱动器间的同步脉冲信号, LCD 驱动器通过它来将水平移位寄存器中的内容显示到 LCD 屏上。LCD 控制器在一整行数据全部传输到 LCD 驱动器后发出 VLINE 信号。
- L\_PCLK\_WR: 此信号为 LCD 控制器和 LCD 驱动器之间的像素时钟信号, LCD 控制器在 VCLK 的上升沿发送数据, LCD 驱动器在 VCLK 的下降沿采样数据。
- L\_BIAS: LCD 驱动器所使用的交流信号。LCD 驱动器使用 VM 信号改变用于打开或关闭像素的行和列电压的极性。VM 信号可在每一帧触发, 也可在数量可编程的一些 VLINE 信号后触发。
- L\_DD[17:0]: LCD 像素数据输出端口。
- L\_CL: LCD 片选信号。

## 5. 其他显示接口原理与结构

### (1) LED 的显示原理

发光二极管(Light Emitting Diode, 简称 LED)常称为七段发光二极管, 在专用的微型计算机系统中, 特别是在嵌入式控制系统中, 应用非常普遍。这种显示器价格低廉、体

积小、功耗低，而可靠性又很好，因此，从单板微型机、袖珍计算机到许多微型机控制系统及数字化仪器都用 LED 作为输出显示。

LED 的主要部分为七段发光管，如图 2-40 (a) 所示。7 个字段分别称为 a、b、c、d、e、f、g 段，有时还有一个小数点段 DP。七段式发光管名称就是由此而来。通过 7 个发光段的不同组合，可以显示 0~9 和 A~F 共 16 个字母数字，从而实现十六进制的显示。例如，控制 a、b、c、d、e、f 段亮 g 段不亮，就显示出数字零。

LED 可以分为共阳极和共阴极两种结构，如图 2-40 (b)、(c) 所示。

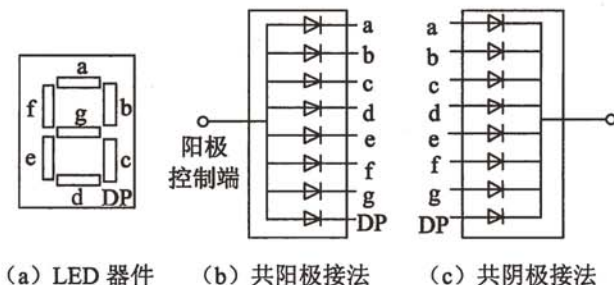


图 2-40 7 段 LED 显示器

当使用共阴极结构时，各字段阴极控制端共接低电平，各字段阳极控制端凡接高电平则该段发光。例如，要显示 b 字母，只要使 c、d、e、f、g 阳极接高电平即可实现。

如为共阳极结构，则数码显示端输入低电平有效，当某一段接低电平时，便发光。比如，当 a、b、g、e、d 为低电平，而其他段为高电平时，则显示数字“2”。

在多个 LED 显示电路中，通常把阴（阳）极控制端接至一输出端口，称它为位控端口；而把数据显示段接至一个输出端口，称这个端口为段控端口。段控端口处应输出十六进制数的 7 段代码。七段 LED 代码如表 2-10 所示。

表 2-10 七段 LED 代码表

	共阴接法								七段 代码	共阳接法								七段 代码
	D7	D6	D5	D4	D3	D2	D1	D0		D7	D6	D5	D4	D3	D2	D1	D0	
	dp	g	f	e	d	c	b	a		dp	g	f	e	d	c	b	a	
0	0	0	1	1	1	1	1	1	3FH	1	1	0	0	0	0	0	0	C0H
1	0	0	0	0	0	1	1	0	06H	1	1	1	1	1	0	0	1	F9H
2	0	1	0	1	1	0	1	1	5BH	1	0	1	0	0	1	0	0	A4H
3	0	1	0	1	1	1	1	1	4FH	1	0	1	1	0	0	0	0	B0H
4	0	1	1	0	0	1	1	0	66H	1	0	0	1	1	0	0	1	99H

续表

	共阴接法								七段 代码	共阳接法								七段 代码
	D7	D6	D5	D4	D3	D2	D1	D0		D7	D6	D5	D4	D3	D2	D1	D0	
	dp	g	f	e	d	c	b	a		dp	g	f	e	d	c	b	a	
5	0	1	1	0	1	1	0	1	6DH	1	0	0	1	0	0	1	0	92H
6	0	1	1	1	1	1	0	1	7DH	1	0	0	0	0	0	1	0	82H
7	0	0	0	0	0	1	1	1	07H	1	1	1	1	1	0	0	0	F8H
8	0	1	1	1	1	1	1	1	7FH	1	0	0	0	0	0	0	0	80H
9	0	1	1	0	1	1	1	1	6FH	1	0	0	1	0	0	0	0	90H
A	0	1	1	1	0	1	1	1	77H	1	0	0	0	1	0	0	0	88H
B	0	1	1	1	1	1	0	0	7CH	1	0	0	0	0	0	1	1	83H
C	0	0	1	1	1	0	0	1	39H	1	1	0	0	0	1	1	0	C6H
D	0	1	0	1	1	1	1	0	5EH	1	0	1	0	0	0	0	1	A1H
E	0	1	1	1	1	0	0	1	79H	1	0	0	0	0	1	1	0	86H
F	0	1	1	1	0	0	0	1	71H	1	0	0	0	1	1	1	0	8EH
P	0	1	1	1	0	0	1	1	73H	1	0	0	0	1	1	0	0	8CH

为了将一个 4 位二进制数（可能为一个十六进制数，也可能是一个 BCD 码）在一个 LED 上显示出来，就需要将 4 位二进制数译为 LED 的 7 位显示代码。要完成译码功能，可以采用两种方法。

- 一种方法是采用专用芯片，如 7447，即采用专用的带驱动器的 LED 段译码器，可以实现对 BCD 码的译码，但不能对大于 9 的二进制数译码。7447 有 4 位输入，7 位输出。使用时，只要将 7447 的输入端与主机系统输出端口的某 4 个数据位（也可以和存储器的某 4 位输出）相连，而 7447 的 7 位输出直接与 LED 的 a 到 g 相接，便可实现对 1 位的 BCD 码的显示。
- 另一种常用的办法是软件译码法。在软件设计时，将 0~F 共 16 个数字（也可以为 0~9）对应的显示代码组成一个表。

## (2) VGA 接口

LCD 虽然可以作为嵌入式产品显示的最佳解决方案，但在某些需要大屏幕显示的应用中，LCD 的价格依然显得比较昂贵。现有的大屏幕显示器(包括 CRT 显示器和 LCD)一般都采用统一的 15 引脚 VGA (Video Graphic Array) 显示接口。

VGA 接口的信号定义如表 2-11 所示，15 引脚的 VGA 接口中除了 2 个 NC 信号、3 根显示器数据总线和 5 个 GND 信号，真正比较重要的信号也就是 3 个 RGB 色彩分量信号和 2 个扫描同步信号 HSYNC 和 VSYNC。

表 2-11 15 引脚的 VGA 接口信号定义

信 号 定 义	信 号 描 述
RED	红色分量的电平信号
GREEN	绿色分量的电平信号
BLUE	蓝色分量的电平信号
HSYNC	行同步扫描信号
VSYNC	帧同步扫描信号
GND	数字地
SYNC GND	扫描信号地
RED GND	红色信号地
GREEN GND	绿色信号地
BLUE GND	蓝色信号地
MONITOR ID [2:0]	显示器数据总线, 用于传递命令和显示器参数

VGA 接口本质上是一个模拟接口, 在数字化的潮流下, 这个接口显然已经比较过时了。近年来, 新的数字化显示接口的标准不断地被提出, 比较有影响的是 DDWG (Digital Display Working Group) 提出的 DVI (Digital Visual Interface) 接口, 不过由于数字接口的标准还未统一, 众多厂家纷纷支持各自的标准, 导致数字接口的标准迟迟没有得到确定。而作为在显示领域多年的行业标准, VGA 接口直到今天也是所有显示设备的标准接口, 而且现在某些高端的电视也支持该接口。

VGA 接口中色彩分量采用的电平标准为 EIA (Electronics Industry Association, 电子工业协会) 定义的 RS343 标准。该标准定义的 4 个电平范围是:

- White +0.714V。
- Black +0.054V。
- Blank 0V。
- Sync -0.286V。

对于普通的 VGA 显示器, 其引出线共含 5 个信号: G、R、B, 三基色信号; HS, 行同步信号; VS, 场同步信号。

对于 5 个信号的时序驱动, 对于 VGA 显示器要严格遵循“VGA 工业标准”, 即 640×480×60Hz 模式。通常用的显示器都满足工业标准, 因此设计 VGA 控制器时要参考显示器的技术规格。图 2-41 是 VGA 行扫描、场扫描的时序图。

VGA 工业标准所要求的频率如下:

- 时钟频率 (Clock frequency) 25.175MHz (像素输出的频率)。
- 行频 (Line frequency) 31469Hz。
- 场频 (Field frequency) 59.94Hz (每秒图像刷新频率)。

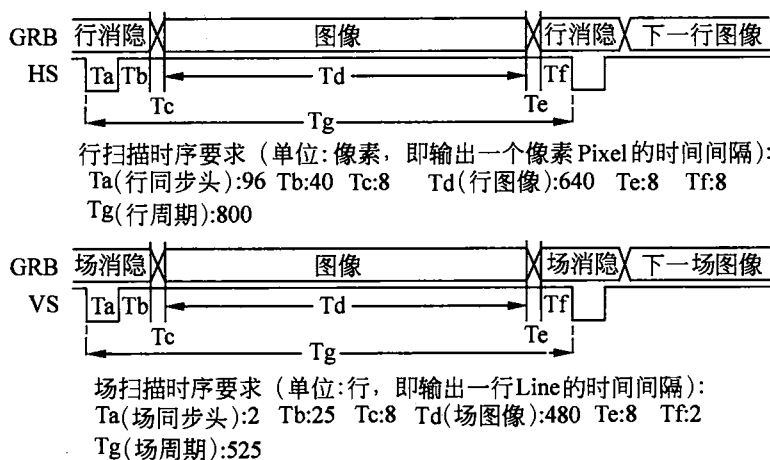


图 2-41 VGA 行扫描、场扫描时序图

## 2.3.7 触摸屏接口基本原理与结构

### 1. 触摸屏原理

触摸屏按其工作原理的不同分为表面声波屏、电容屏、电阻屏和红外屏几种。而常见的又数电阻触摸屏。如图 2-42 所示，电阻触摸屏的屏体部分是一块与显示器表面非常配合的多层复合薄膜，由一层玻璃或有机玻璃作为基层，表面涂有一层透明的导电层，上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层透明导电层，在两层导电层之间有许多细小（小于千分之一英寸）的透明隔离点把它们隔开绝缘。

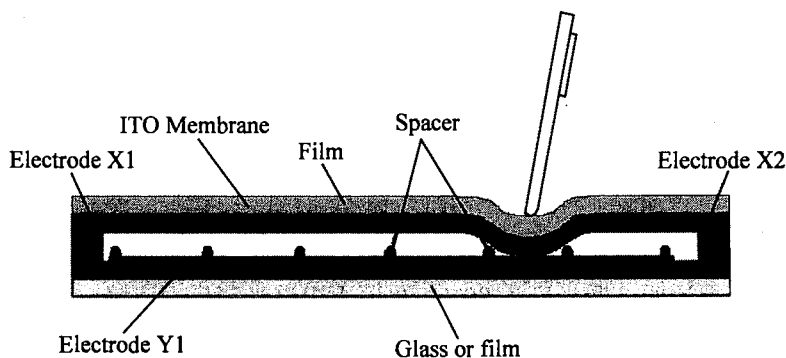


图 2-42 触摸屏的结构

当手指或笔触摸屏幕时（如图 2-43（c）所示），平常相互绝缘的两层导电层就在触摸



点位置有了一个接触，因其中一面导电层（顶层）接通  $X$  轴方向的  $5V$  均匀电压场（如图 2-43（a）所示），使得检测层（底层）的电压由零变为非零，控制器侦测到这个接通后，进行 A/D 转换，并将得到的电压值与  $5V$  相比即可得触摸点的  $X$  轴坐标为（原点在靠近接地的那端）：

$$X_i = L_x * V_i / V \quad (\text{即分压原理})$$

同理得出  $Y$  轴的坐标，这就是所有电阻技术触摸屏共同的最基本原理。

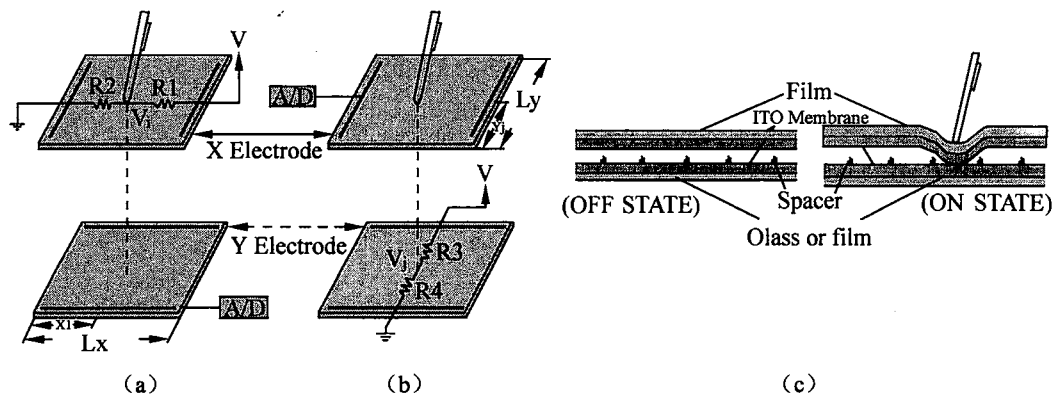


图 2-43 触摸屏坐标识别原理

## 2. 电阻触摸屏的有关技术

电阻触摸屏的主要部分是一块与显示器表面非常配合的电阻薄膜屏，这是一种多层的复合薄膜，由一层玻璃或有机玻璃作为基层，表面涂有一层叫作 ITO 的透明导电层，上面再盖有一层外表面硬化处理、光滑防刮的塑料层，它的内表面也涂有一层导电层（ITO 或镍金）。电阻触摸屏的两层 ITO 工作面必须是完整的，在每个工作面的两条边线上各涂一条银胶，一端加  $5V$  电压，一端加  $0V$ ，就能在工作面的一个方向上形成均匀连续的平行电压分布。在侦测到有触摸后，立刻 A/D 转换测量接触点的模拟量电压值，根据  $5V$  电压下的等比例公式就能计算出触摸点在这个方向上的位置。

透明的导电涂层有两种：

- ITO，氧化锡，弱导体，特性是当厚度降到 1800 个埃（1 埃 =  $10^{-10}$  米）以下时会突然变得透明，透光度为 80%，再薄下去透光率反而下降，到 300 埃厚度时又上升到 80%。但有遗憾是 ITO 在这个厚度下非常脆，容易折断产生裂纹。ITO 是所有电阻技术触摸屏及电容技术触摸屏都用到的主要材料，实际上电阻和电容技术触摸屏的工作面就是 ITO 涂层。
- 镍金涂层，五线电阻触摸屏的外层导电层使用的是延展性极好的镍金涂层材料，

外导电层由于频繁触摸,使用延展性好的镍金材料目的是为了延长使用寿命,但是成本较为高昂,镍金导电层虽然延展性好,但是只能作透明导体,不适合作为电阻触摸屏的工作面,因为它导电性太好,不直接作精密电阻测量,而且金属不易做到厚度非常均匀。

第一代四线触摸屏两层 ITO 工作面工作时都加上 5V~0V 的均匀电压分布场:一个工作面加竖直方向的,一个工作面加水平方向的。引线至控制器总共需要 4 根电缆。因为四线电阻触摸屏靠外的那层塑胶及 ITO 涂层被经常触动,一段时间后外层薄薄的 ITO 涂层就有了细小的裂纹。显然,导电工作面一旦有了裂纹,电流就会绕之而过,工作面上的电压场分布也就不可能再均匀。这样,在裂纹附近触摸屏漂移严重,裂纹增多后,触摸屏有些区域可能就再也触摸不到了。

四线电阻触摸屏的基层大多数是有机玻璃,不仅存在透光率低、风化、老化的问题,并且存在安装风险,这是因为有机玻璃刚性差,安装时不能捏边上的银胶,以免薄薄的 ITO 和相对厚实的银胶脱裂,不能用力压或拉触摸屏,以免拉断 ITO 层。有些四线电阻触摸屏安装后显得不太平整就是这个原因。

ITO 是无机物,有机玻璃是有机物,有机物和无机物是不能良好结合的,时间一长就容易剥落。如果能够生产出曲面的玻璃板(玻璃是无机物,能和 ITO 非常好的结合为导电玻璃)电阻触摸屏的寿命能够大大延长。

第二代五线电阻技术触摸屏的基层使用的就是这种导电玻璃,不仅如此,五线电阻技术把两个方向的电压场通过精密电阻网络都加在玻璃的导电工作面上,可以简单地理解为两个方向的电压场分时加在同一工作面上,而外层镍金导电层只仅仅用来当作纯导体,有触摸后靠既检测内层 ITO 接触点电压又检测导通电流的方法测得触摸点的位置。五线电阻触摸屏内层 ITO 需 4 条引线,外层只作导体仅仅一条,至控制器总共需要 5 根电缆。因为五线电阻屏的外层镍金导电层不仅延展性好,而且只作导体,只要它不断成两半,就仍能继续完成作为导体的使命,而身负重任的内层 ITO 直接与基层玻璃结合为一体成为导电玻璃,导电玻璃自然没有了有机玻璃作基层的种种弊端,因此,五线电阻屏的使用寿命和透光率与四线电阻屏相比有了一个飞跃:五线电阻屏的触摸寿命是 3500 万次,四线电阻屏则是小于 100 万次,且五线电阻触摸屏没有安装风险,同时五线电阻屏的 ITO 层能做得更薄,因此透光率和清晰度更高,几乎没有色彩失真。

不管是四线电阻触摸屏还是五线电阻触摸屏,它们都是一种对外界完全隔离的工作环境,不怕灰尘、水汽和油污,它可以用任何物体来触摸,可以用来写字画画,比较适合工业控制领域使用。电阻触摸屏共同的缺点是因为复合薄膜的外层采用塑胶材料,不知道的人太用力或使用锐器触摸可能划伤整个触摸屏而导致报废。不过,在限度之内,划伤只会伤及外导电层,外导电层的划伤对于五线电阻触摸屏来说没有关系,而对四线电阻触摸屏

来说是致命的。

### 3. 触摸屏的控制

触摸屏的控制采用专用芯片，专门处理是否有笔或手指按下触摸屏，并在按下时分别给两组电极通电，然后将其对应位置的模拟电压信号经过 A/D 转换送回处理器。触摸屏的控制结构如图 2-44 所示。

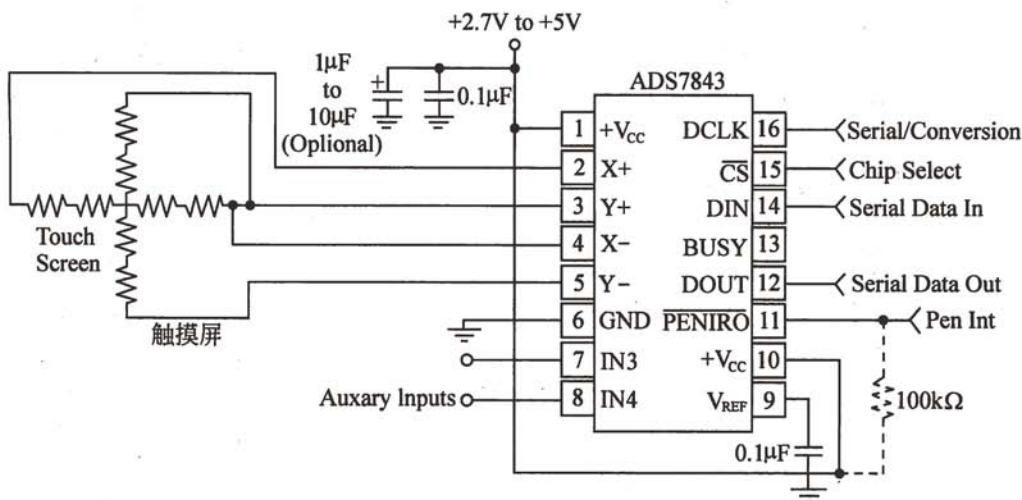


图 2-44 触摸屏控制结构 (ADS7843)

### 4. 触摸屏与显示器的配合

ADS7843 送回控制器的  $X$  与  $Y$  值仅是对当前触摸点的电压值的 A/D 转换值，它不具有实用价值。这个值的大小不但与触摸屏的分辨率有关，而且也与触摸屏与 LCD 贴合的情况有关。而且，LCD 分辨率与触摸屏的分辨率一般来说不一样，坐标也不一样，因此，如果想得到体现 LCD 坐标的触摸屏位置，还需要在程序中进行转换。假设 LCD 分辨率是  $320 \times 240$ ，坐标原点在左上角；触摸屏分辨率是  $900 \times 900$ ，坐标原点在左上角，则转换公式如下：

$$x_{LCD} = [320 \times (x - x_2) / (x_1 - x_2)];$$

$$y_{LCD} = [240 \times (y - y_2) / (y_1 - y_2)];$$

如果坐标原点不一致，比如 LCD 坐标原点在右下角，而触摸屏原点在左上角，则还可以进行如下转换：

$$x_{LCD} = 320 - [320 \times (x - x_2) / (x_1 - x_2)];$$

$$y_{LCD} = 240 - [240 \times (y - y_2) / (y_1 - y_2)];$$

最后得到的值,便可以尽可能使 LCD 坐标与触摸屏坐标一致,这样,更具有实际意义。

### 2.3.8 音频接口基本原理与结构

目前,越来越多的嵌入式系统产品,如 CD、手机、MP3、MD、VCD、DVD、数字电视等,引入了数字音频系统。这些产品中数字化的声音信号由一系列的超大规模集成电路处理,常用的数字声音处理需要的集成电路包括 A/D 转换器和 D/A 转换器、数字信号处理器(DSP)、数字滤波器和数字音频输入输出接口及设备(麦克风、话筒)等。麦克风输入的数据经音频编解码器解码完成 A/D 转换。解码后的音频数据送入通过音频控制器送入 DSP 或 CPU 进行相应的处理。音频输出数据经音频控制器发送给音频编码器,经编码 D/A 转换后由扬声器输出。

#### 1. 音频数据类型

数字音频数据有多种不同格式。下面简要介绍 3 种最常用的格式:采样数字音频(PCM)、MPEG 层 3 音频(MP3)和 ATSC 数字音频压缩标准(AC3)。

- PCM 数字音频是 CD-ROM 或 DVD 采用的数据格式。对左右声道的音频信号采样得到 PCM 数字信号,采样率为 44.1kHz,精度为 16 位或 32 位。因此,精度为 16 位时,PCM 音频数据速率为 1.41Mb/s; 32 位时为 2.42Mb/s。一张 700MB 的 CD 可保存大约 60 分钟的 16 位 PCM 数据格式的音乐。
- MP3 是 MP3 播放器采用的音频格式,对 PCM 音频数据进行压缩编码。立体声 MP3 数据速率为 112kb/s 至 128kb/s。对于这种数据速率,解码后的 MP3 声音效果与 CD 数字音频的质量相同。
- AC3 是数字 TV、HDTV 和电影数字音频编码标准。立体声 AC3 编码后的数据速率为 192kb/s。

编码或解码音频数据的常用串行音频数字接口是 Inter-IC 音频总线(IIS)。每个音频字的边界由信号 WS 标识。在应用中采用配置模式一。在 SCK 信号的上升沿,数据被锁存至接收器,但是当 SCK 保持低电平时,不接收数据。

#### 2. IIS 音频接口总线

##### 1) IIS 总线简介

数字音频系统需要多种集成电路,所以为这些电路提供一个标准的通信协议非常重要。IIS 总线是 Philips 公司提出的音频总线协议,全称是数字音频集成电路通信总线(Inter-IC Sound Bus, IIS),它是一种串行的数字音频总线协议。音频数据的编码或解码的常用串行音频数字接口是 IIS 总线。

IIS 总线只处理声音数据,其他控制信号等则需单独传输。IIS 使用了 3 根串行总线,以尽量减少引出管脚,这 3 根线分别是:提供分时复用功能的数据线;字段选择线(声道

选择); 时钟信号线。

数据的发送方和接收方需要有相同的时钟信号来控制数据传输, 所以数据传输方(主设备)必须产生字段选择信号、时钟信号和需要传输的数据信号。复杂的数字音频系统可能会有多个发送方和接收方, 因此很难定义哪个是主设备。这种系统一般会有一个系统主控制模式, 用于控制数字音频数据在不同集成电路间的传输。引入主控制模块后, 数据发送方就需要在主控制模块的协调下发送数据。图 2-45 为几种传输模式, 这些模式的配置一般需通过软件来实现。

## 2) IIS 总线协议

IIS 总线的时刻状态图如图 2-46 所示, 其中说明了 IIS 中时钟信号、字段选择和串行数据传输信号之间的同步关系。

### (1) 串行数据传输 (SD)

串行数据的传输由时钟信号同步控制, 且串行数据线上每次传输一个字节的数据。当音频数据被数字化成二进制流后, 传输时先将数据分成字节, 如 8 位、16 位等, 每个字节的数据传输从左边的二进制位 MSB (Most Significant Bit) 开始。当接收方和发送方的数据字段宽度不一样时, 发送方不考虑接收方的数据字段宽度。

如果发送方发送的数据字段宽度小于系统字段宽度, 就在低位补 0; 如果发送方的数据宽度大于接收方的宽度, 则超过 LSB (Least Significant Bit) 的部分被截断。

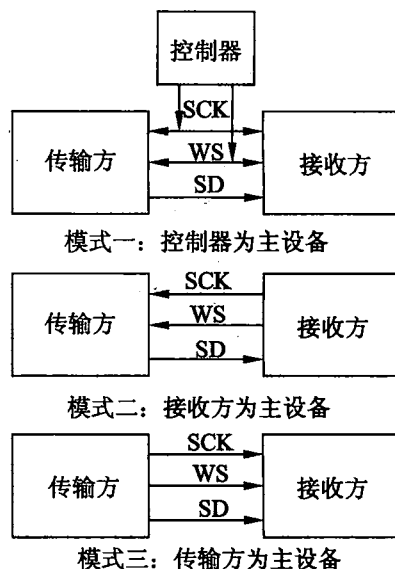


图 2-45 IIS 数据传输格式

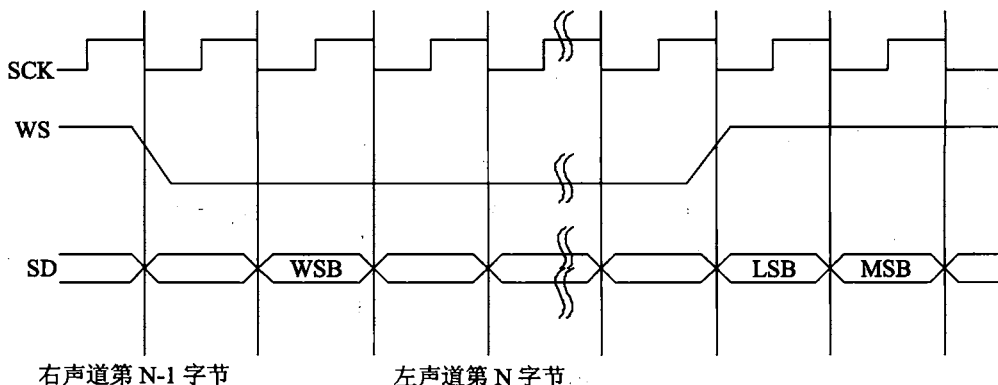


图 2-46 IIS 的数据传输

### (2) 字段选择 (WS)

音频一般由左声道和右声道组成,使用字段选择就是用来选择左右声道,WS=0 表示选择左声道;WS=1 表示选择右声道。如果不再外部加以控制,WS 会在 MSB 传输前的一个时钟周期发生变化,这有助于数据接收方和发送方保持同步。此外,WS 能让接收设备存储前一个字节,并且准备接收下一个字节。

### (3) 时钟信号 (SCK)

IIS 总线中,任何一个能够产生时钟信号的集成电路都可以称为主设备,从设备从外部时钟的输入得到时钟信号。IIS 的规范中制定了一系列关于时钟信号频率和延时的限制。

## 2.4 嵌入式系统总线接口

### 2.4.1 串行接口基本原理与结构

#### 1. 串行通信概述

所谓串行通信就是使数据一位一位地进行传输而实现的通信。当然,在实际传输中,如外部设备与 CPU 或计算机与计算机之间交换信息,是通过一对导线传送信息的。在传输中每一位数据都占据一个固定的时间长度。与并行通信相比,串行通信具有传输线少、成本低等优点,特别适合远距离传送,其缺点是速度慢,若并行传送  $n$  位数据需时间  $T$ ,则串行传送的时间最少为  $nT$ 。

##### 1) 串行数据传送模式

数据通信涉及两台数字设备之间传输数据的问题。常用的数据通信方式有并行通信和串行通信两种。当距离较近而且要求传输速率较高时,通常采用并行通信的方式,计算机系统的内部总线结构就是并行方式。当设备距离较远时,数据往往以串行方式传输。图 2-47 列出了 3 种基本的通信模式。

- 单工通信:数据仅能沿着从 A 到 B 的单一方向传播。
- 半双工通信:数据可以从 A 到 B,也可以从 B 到 A,但不能在同一时刻传播。
- 全双工通信:数据在同一时刻可以从 A 到 B,或从 B 到 A 进行双向传播。

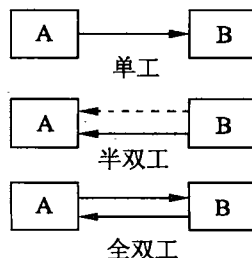


图 2-47 3 种基本的通信模式

##### 2) 串行通信方式

串行通信在信息格式的约定上可以分为两种方式:同步通信和异步通信。

##### (1) 异步通信方式

异步式传输把每一个字符当作独立的信息来传送,并按照一固定且预定的时序传

送，但在字符之间却取决于字符与字符的任意时序。而一个完整的字符传送，包含一个起始位以及所欲传送的字符，加上校验位和停止位。以下将说明单个字节经异步传输的位时序。

当一个字符要传送到某接收器时，是以其最低有效位 (LSB) 先送出的 (即  $D_0$ )，但为使接收器能事先知道开始传送，所以先使串行通信数据线在无数据传送时都固定保持在一个状态上。假设无数据在串接数据线上时，其状态固定保持在“1”，称此数据线在空闲状态，而为使接收器知道数据开始传送，所以在传送第一个位 ( $D_0$ ) 时，先传送一个与空闲状态相反的状态，即状态“0”，当作起始位，如此当串行数据线由空闲状态“1”转变到所传送的起始位“0”，接收器就能通过检测状态的变化而知道数据的开始传送。所以假设有如下的位串：

$D_7$	$D_6$	$D_5$	$D_4$	$D_3$	$D_2$	$D_1$	$D_0$
0	0	0	1	0	0	1	1

则欲传送该位串的字符时，其顺序即为先传送起始位“0”，再接着传送“11001000”，而当传送完  $D_7$  后，可以再接着传送一个奇偶校验位，用来进行错误检测。最后发送至少一个停止位“1”，以区分下一个字符的起始位“0”。这样构成的一串数据称为一帧。一帧数据的各位代码间的时间间隔是固定的，而相邻两帧的数据其时间间隔是不固定的。可见，异步通信时字符是一帧一帧传送的，每帧字符的传送靠起始位来同步。在异步通信的数据传送中，传输线上允许空字符。

异步通信必须遵循的 3 项规定为：

- 字符的格式

每个字符传送时，必须前面加一起始位，后面加上 1、1.5 或 2 位停止位。例如 ASCII 码传送时，一帧应该是：前面 1 个起始位，接着 7 位 ASCII 编码，再接着一位奇偶校验位，最后一位停止位，共 10 位。

- 波特率

波特率就是传送数据位的速率，用位/秒 (bit/s) 表示，称之为波特。例如，数据传送的速率为 120 字符/秒，每帧包括 10 个数据位，则传送波特率为：

$$10 \times 120 = 1200 \text{ b/s} = 1200 \text{ 波特}$$

每一位的传送时间是其倒数  $1/1200 = 0.833 \text{ ms}$ 。一般情况下，异步通信的波特率的值为：150、300、600、1200、2400、4800、9600、14400、28800 等，数值成倍数变动，这是因为采用一个基准时序再做二次方分频后的结果。

- 校验位

在一个有 8 位的字节 (byte) 中，其中必有奇数个或偶数个的状态“1”位。假设接收

器的硬件设计要接收偶数个“1”，无论起始位和停止位，当字符内有偶数个“1”时，则校验位就设为“1”，反之，字符内有奇数个“1”时，则其设为“0”。换言之，对于偶校验就是要使字符加上校验位有偶数个“1”；奇校验就是要使字符加上校验位有奇数个“1”。例如在上例中，共有奇数个“1”，所以当接收器要接收偶数个“1”时（即偶校验），则校验位就置为“1”，反之，接收器要接收奇数个“1”时，则校验位就置为“0”。

一般校验位的产生和检查是由串行通信控制器内部自动产生，除了加上校验位以外，通信控制器还自动加上停止位，用来指明欲传送字符的结束。对接收器而言，若未能检测到停止位则意味着传送过程发生了错误。而停止位会根据计算机的种类取 1、1.5 或 2 个位。

## （2）同步通信方式

在前述的异步通信方式中，可以看到在发送的数据中含有起始位和停止位这两个与实际欲传送的数据毫无相关的位。换句话说，若在传送 1 个 8 位的字符时，其校验位、起始位和停止位都为 1 个位，则相当于要传送 11 个位信号，所以实际上的使用率就只有约 80% 而已。显然当需要高速率的通信速度时，异步式的传输不能满足需求。因此，为了提高通信效率就提出了同步通信方式。

与异步方式不同的是，同步方式不仅在字符的本身之间是同步的，而且在字符与字符之间的时序仍然是同步的，即同步方式是将许多的字符聚集成一字符块后，在每块信息（常常称之为信息帧）之前要加上 1~2 个同步字符，字符块之后再加入适当的错误检测数据才传送出去。采用同步通信时，在传输线上没有字符传输时，要发送专用的“空闲”字符或同步字符，其原因是同步传输字符必须连续传输，不允许有间隙。

由于同步传输采用字符块的方式，所以相对于异步方式里每一字符就有一对控制数据和错误检测数据的设计，同步方式的字符块中的每一个字符就有比较少的控制数据和错误检测数据，因而有较高的传输速率。更重要的是，异步方式下虽然有校验位可用以检测错误，但其功能也只能检测错误，而不能进行任何的修正操作，而且对于偶数个错误位的产生就不易检测出来。

在同步方式中产生一种所谓“冗余”字符，使其有较高的错误防止率。这种“冗余”字符的含义即为，假设欲传送的数据位当作一被除数，而发送器本身产生一固定的除数，将前者除以后者所得的余数即为该“冗余”字符。当数据位和“冗余”字符位一起被传送到接收器时，接收器产生和发送器相同的除数，如此即可检查出数据在传送过程中是否发生了错误。此法不但可防止奇数个或偶数个错误的发生，而且经过统计的数据表明错误防止率可达 99% 以上。

## 2. RS-232C 串行接口

RS-232C 是由美国电子工业协会 EIA 于 1969 年制定并采用的一种串行通信接口标准，



后来被广泛采用，发展成为一种国际通用的串行通信接口标准。

### (1) RS-232C 接口规格

表 2-12 所示为 EIA 所定的传送电气规格。

表 2-12 EIA 的所定的传送电气规格

状 态	L(Low)	H(Hight)
电压范围	-25V~-3V	+3V~+25V
逻辑	1	0
名称	SPACE	MARK

RS-232C 所用的驱动芯片通常以 $\pm 12\text{V}$  的电源来驱动信号线，但是实际上，因为传输线的连接状态及接收端负载阻抗的影响，均会造成电压的下降，但最低仍不得低于 $\pm 5\text{V}$  以下。

通常，微机系统以 $+5\text{V}$  代表逻辑“1”，而接地电压代表逻辑“0”，输出经 RS-232C 接口内的运算放大器改变为 $\pm 12\text{V}$  的振幅电压，再由电线传送到接收端。因为数据信号规定负逻辑，所以 $+12\text{V}$  的电压经接收端的运算放大器后调整为 $0\text{V}$  电压而被视为逻辑“0”。相反， $-12\text{V}$  的电压经转换后被视为逻辑“1”。但整体来说，两个端点的逻辑仍为一致的，TTL 标准与 RS-232C 标准之间的电平转换电路利用集成芯片 RS232 实现，如图 2-48 所示。

### (2) RS-232C 接口信号

一般常见到的 RS-232C 接口接头大都以 D 型 25 引脚的连接座 (DB-25) 与外界相连，表 2-13 列出了引脚和信号之间的对应关系。实际应用中，并不是每只引脚信号都必须用到，所以当初 IBM 公司在开发自己的系统时又将其缩减为只有 9 线，在微机中又用了 9 芯 D 型连接器 (如图 2-49 所示)，其引脚信号规定见表 2-13 所示。

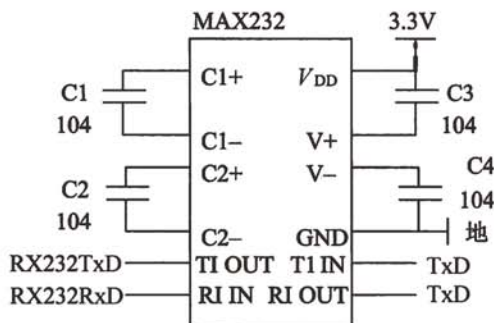


图 2-48 RS-232C 和 TTL 之间的电平转换

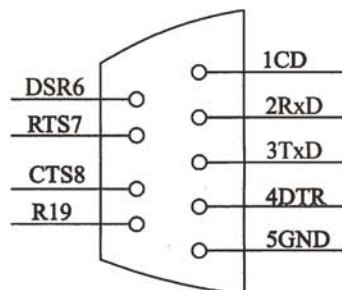


图 2-49 9 芯 D 型连接器信号规定

表 2-13 RS-232C 的信号定义表

引脚号	符号表示	名称	说明
1	FG	保护地	作为设备接地端
2	TxD	发送数据(出)	将数据送调制解调器(Modem)
3	RxD	接收数据(入)	从 Modem 接收数据
4	$\overline{\text{RTS}}$	请求发送	在半双工方式下控制发送器的开或关
5	$\overline{\text{CTS}}$	允许发送	指出 Modem 准备好发送
6	$\overline{\text{DSR}}$	数据装置准备好	指出 Modem 可进入工作状态
7	GND	信号地	作为所有信号的公用地
8	DCD	载波信号检测	指出 Modem 正在接收另一端送来的信号
9	空		
10	空		
11	空		
12		第二通道接收信号检测	指出在第二通道上检测到信号
13		第二通道允许发送	指出第二通道准备发送
14		第二通道发送数据	往 Modem 以较低速率输出
15		发送器定时	Modem 提供发送器定时信号
16		第二通道接收数据	从 Modem 以较低速率输入
17		接收器定时	为接口和终端接收器提供信号定时
18	空		
19		第二通道请求发送	闭合第二通道的发送器
20	$\overline{\text{DTR}}$	数据终端准备好	Modem 连接到链路, 并开始发送
21	空		
22	RI	音响指示	指出在链路上检测到音响信号
23		数据率选择	可选择两个同步数据率之一
24		发送器定时	为接口和终端提供发送器定时信号
25	空		

### 3. RS-422 串行通信接口

RS-422 由 RS-232 发展而来。为改进 RS-232 通信距离短、速度低的缺点, RS-422 定义了一种平衡通信接口, 将传输速率提高到 10Mb/s, 允许在一条平衡总线上连接最多 10 个接收器。RS-422 是一种单机发送、多机接收的单向、平衡传输规范。RS-422 的数据信号采用差分传输方式, 也称做平衡传输。它使用一对双绞线进行数据传输。

RS-422 标准全称是“平衡电压数字接口电路的电气特性”。由于接收器采用高输入阻

抗并且发送驱动器具有比 RS-232 更强的驱动能力, 故允许在相同传输线上连接多个接收节点, 最多可接 10 个节点, 即一个主设备 (Master), 其余为从设备 (Slave), 从设备之间不能通信, 所以 RS-422 支持点对多的双向通信。RS-422 四线接口由于采用单独的发送和接收通道, 因此不必控制数据方向, 各装置之间任何必需的信号交换均可以按软件方式 (XON/XOFF 握手) 或硬件方式 (一对单独的双绞线) 实现。

RS-422 的最大传输距离为 4000 英尺 (约 1219m), 最大传输速率为 10Mb/s。其平衡双绞线的长度与传输速率成反比, 在 100Kb/s 速率以下, 才可能达到最大传输距离。只有在很短的距离下才能获得最高传输速率。一般 100m 长的双绞线上所能获得的最大传输速率仅为 1Mb/s。

RS-422 需要一终接电阻, 要求其阻值约等于传输电缆的特性阻抗。在短距离传输时可不需终接电阻, 即一般在 300m 以下不需终接电阻。终接电阻接在传输电缆的最远端。

#### 4. RS-485 串行总线接口

为扩展应用范围, EIA 在 RS-422 的基础上制定了 RS-485 标准, 增加了多点、双向通信能力, 通常在要求通信距离为几十米至上千米时, 广泛采用 RS-485 收发器。RS-485 收发器采用平衡发送和差分接收, 即在发送端, 驱动器将 TTL 电平信号转换成差分信号输出; 在接收端, 接收器将差分信号变成 TTL 电平, 因此具有抑制共模干扰的能力, 加上接收器具有高的灵敏度, 能检测低达 200mV 的电压, 故数据传输可达千米以外。

RS-485 许多电气规定与 RS-422 相仿, 如都采用平衡传输方式, 都需要在传输线上接终端电阻等。RS-485 可以采用二线与四线方式, 二线制可实现真正的多点双向通信。而采用四线连接时, 与 RS-422 一样只能实现点对多的通信, 即只能有一个主设备, 其余为从设备, 但它比 RS-422 有所改进。无论四线还是二线连接方式总线上都可连接多达 32 个设备。

RS-485 与 RS-422 的共模输出电压是不同的。RS-485 共模输出电压在  $-7V \sim +12V$  之间, RS-422 在  $-7V \sim +7V$  之间, RS-485 接收器最小输入阻抗为  $12k\Omega$ ; RS-422 是  $4k\Omega$ ; RS-485 满足所有 RS-422 的规范, 所以 RS-485 的驱动器可以在 RS-422 网络中应用, 但 RS-422 的驱动器并不完全适用于 RS-485 网络。

RS-485 与 RS-422 一样, 最大传输速率为 10Mb/s。当波特率为 1200b/s 时, 最大传输距离理论上可达 15km。平衡双绞线的长度与传输速率成反比, 在 100kb/s 速率以下, 才可能使用规定最长的电缆长度。

RS-485 需要两个终端电阻, 接在传输总线的两端, 其阻值要求等于传输电缆的特性阻抗。在短距离传输时可不需终端电阻, 即一般在 300m 以下不需终端电阻。

表 2-14 给出了 RS-232、RS-422 和 RS-485 之间的性能比较。

表 2-14 RS-232/422/485 接口电路特性比较

规 定	RS-232	RS-422	RS-485
工作方式	单端	差分	差分
节点数	1 收、1 发	1 发 10 收	1 发 32 收
最大传输电缆长度	50inch	400inch	400inch
最大传输速率	20Kb/S	10Mb/s	10Mb/s
最大驱动输出电压	+/-25V	-0.25V~+6V	-7V~+12V
驱动器输出信号电平 (负载最小值)	+/-5V~+/-15V	+/-2.0V	+/-1.5V
驱动器输出信号电平 (空载最大值)	+/-25V	+/-6V	+/-6V
驱动器负载阻抗 ( $\Omega$ )	3k~7k	100	54
摆率 (最大值)	30V/ $\mu$ s	N/A	N/A
接收器输入电压范围	+/-15V	-10V~+10V	-7V~+12V
接收器输入门限	+/-3V	+/-200mV	+/-200mV
接收器输入电阻 ( $\Omega$ )	3k~7k	4k (最小)	$\geq 12k$
驱动器共模电压		-3V~+3V	-1V~+3V
接收器共模电压		-7V~+7V	-7V~+12V

## 2.4.2 并行接口基本原理与结构

并行接口, 简称并口, 也就是 LPT 接口, 是采用并行通信协议的扩展接口。并行接口的数据传输率比串行接口快 8 倍, 标准并行接口的数据传输率为 1Mb/s, 一般用来连接打印机、扫描仪等。所以并口又被称为打印口。

### 1. 并行接口的分类

并行接口可以分为 SPP (标准并行接口)、EPP (增强型并行接口) 和 ECP (扩展型并行接口)。

#### (1) SPP (标准并行接口)

SPP 也是最早的接口定义, 主要功能如下:

- 并行接口提供了 8 个数据线以进行并行的字节传输。
- 计算机能够通过数据线向打印机发送选能信号, 以通知打印机已经准备好接收数据。
- 打印机接收到数据后, 向计算机发送一个回应信号 (NACK)。

其各位信号线所代表的意义如表 2-15 所示。

表 2-15 并行接口引脚功能

引 脚	功 能	引 脚	功 能
1	选通端，低电平有效	10	确认，低电平有效
2	数据通道 0	11	忙
3	数据通道 1	12	缺纸
4	数据通道 2	13	选择
5	数据通道 3	14	自动换行，低电平有效
6	数据通道 4	15	错误，低电平有效
7	数据通道 5	16	初始化，低电平有效
8	数据通道 6	17	选择输入，低电平有效
9	数据通道 7	18~25	地线

(2) EPP（增强型并行接口）

EPP 的出现提供了一种更高性能的连接方式，并向下兼容所有在此之前存在的并行接口及外设。与 SPP 不同之处在于原来 17 个信号中的部分重新定义，在这 17 个信号中，EPP 使用了其中的 14 个信号进行传输、握手和选通，剩下的 3 个信号可以由外设设计者自定义。

2. 并行总线

并行总线分为标准的和非标准两类。常用的并行标准总线有通用接口总线 IEEE 488 和 ANSI X3.131-1986 SCSI 总线。非标准的并行总线也很多。多数厂家自己设计专用的并行总线，再通过总线转换接口，将计算机和测试部分连接起来。

(1) IEEE 488 总线

IEEE 488 通用接口总线又称 GPIB（General Purpose Interface Bus）总线，是 HP 公司在 20 世纪 70 年代推出的台式仪器接口总线，因此又叫 HPIB（HP Interface Bus），1975 年 IEEE 和 IEC 确认为 IEEE 488 和 IEC 652 标准。IEEE-488 总线是并行总线接口标准。IEEE-488 总线用来连接系统，PC 机、数字电压表、数码显示器等设备及其他仪器仪表均可用 IEEE-488 总线装配起来。它按照位并行、字节串行双向异步方式传输信号，连接方式为总线方式，仪器设备直接并联于总线上而不需中介单元，但总线上最多可连接 15 台设备。最大传输距离为 20m，信号传输速度一般为 500KB/s，最大传输速度为 1MB/s。

该标准总线在仪器、仪表及测控领域得到了最广泛的应用。这种系统是在微机中插入一块 GPIB 接口卡，通过 24 或 25 线电缆连接到仪器端的 GPIB 接口。当微机的总线变化时（如采用 ISA 或 PCI 等不同总线），接口卡也随之变更，其余部分可保持不变，从而使 GPIB 系统能适应微机总线的快速变化。由于 GPIB 系统在 PC 出现的初期问世，所以有一

定的局限性。例如其数据线只有 8 根, 传输速率最高 1Mb/s, 传输距离 20m (加驱动器可达 500m) 等。尽管如此, 目前仍是仪器、仪表及测控系统与计算机互连的主流并行总线。因为装有 GPIB 接口的台式仪器的品种和数量都明显超过备受青睐的 VXI 仪器, 而且在目前应用的 VXI 系统中, 与 GPIB 混合应用比例很大, 还有相当数量采用外主控计算机控制的 VXI 系统, 其计算机通过 GPIB 电缆和 GPIB-VXI 接口进行控制。以 PCI 为基础的 PXI 系统, 也都具有 GPIB 接口。所以, 在相当长的时间内, GPIB 系统仍将在实际应用中, 特别是中、低速范围内的计算机外设总线应用中占有一定的市场。

### (2) SCSI 总线

SCSI 总线的原型是美国 Shugart 公司推出的, 用于计算机与硬盘驱动器之间传输数据的 SASI(Shugart Associates System Interface)总线, 1986 年成为美国国家标准 ANSI X3.131, 改名为 SCSI 总线 (Small Computer System Interface)。其数据线为 9 位, 速度可达 5Mb/s, 传输距离 6m (加驱动器可达 25m), 经改进又陆续推出 SCSI-2 Fast and Wide 和 SCSI-3 (又称 Ultra SCSI) 总线, 原 SCSI 总线改称 SCSI-1 总线。该总线的传输速率很高, 现已普遍用作计算机的高速外设总线, 如连接高速硬盘驱动器。许多高速数据采集系统也用它与计算机互连, 目前仍处在发展之中。

### (3) MXI 总线

MXI (Multi-system eXtension Interface bus, 多系统扩展接口总线) 是一种高性能非标准的通用多用户并行总线, 具有很好的应用前景。它是 NI (National Instruments) 公司于 1989 年推出的 32 位高速并行互连总线, 最高速度可达 23Mb/s, 传输距离 20m。MXI 总线通过电缆与多个器件连接, 采用硬件映象通信设计, 不需要高级软件, 一根 MXI 电缆上可连接 8 个 MXI 器件。其电缆本身是相通的, MXI 器件通过简单地读写相应的地址空间就可直接访问其他所有器件的资源而无需任何软件协议。目前, VXI 总线的测控机箱大都用这种总线与计算机互连。它将成为 VXI 总线机箱与计算机互连的事实上的标准总线。

## 2.4.3 PCI 接口基本原理与结构

在很多低功率的手持嵌入式系统中, 基本的 I/O 设备是与主 CPU 集成在一起的, 不需要主 CPU 总线扩展。但大多数新的设计不仅需要基本的 I/O 设备, 而且很多都采用广泛应用微机系统普遍采用的 PCI (Peripheral Component Interconnect) 总线, 以便于系统的扩展。

PCI 总线是当前最流行的总线之一, 它是由 Intel 公司推出的一种局部总线。1991 年 Intel 公司联合世界上多家公司成立的 PCISIG (Peripheral Component Interconnect Special Interest Group) 协会, 是国际上微型计算机界的行业协会。它致力于促进 PCI 总线工业标准的发

展。PCI 总线规范先后经历了 1.0 版、2.0 版和 1995 年的 2.1 版。PCI 总线是地址、数据多路复用的高性能 32 位和 64 位总线，是微处理器与外围控制部件、外围附加板之间的互连机构。它制定了互连的协议、电气、机械及配置空间规范，以保证全系统的自动配置；在电气方面还专门定义了 5V 和 3.3V 信号、环境，特别是 2.1 版本定义了 64 位总线扩展以及 66MHz 总线时钟的技术规范。

从数据宽度上看，PCI 定义了 32 位数据总线，且可扩展为 64 位。从总线速度上分，有 33MHz 和 66MHz 两种。目前流行的是 32 位@33 MHz，而 64 位系统正在普及中。改良的 PCI 系统 PCI-X，最高可以达到 64 位@133 MHz 的数据传输速度。PCI 总线主板插槽的体积比原 ISA 总线插槽小，其功能比 VESA、ISA 有极大的改善，支持突发读写操作，可同时支持多组外围设备。

与 ISA 总线不同，PCI 总线的地址总线与数据总线是分时复用的，支持即插即用 (plug and plug)、中断共享等功能。分时复用的好处是一方面可以节省介绍接插件的引脚数，另一方面便于实现突发数据传输。

数据传输时，由一个 PCI 设备做发起者（称为 Master、Initiator 或 Master），而另一个 PCI 设备做目标（称为 Slave、Target 或 Slave）。总线上所有时序的产生与控制都由 Master 发起。PCI 总线在同一时刻只能供一对设备完成传输。这就要求有一个仲裁机构来决定谁有权拿到总线的主控权。

32 位 PCI 系统的引脚按功能来分有以下几类。

#### (1) 系统控制

- CLK PCI 时钟，上升沿有效。
- RST Reset 信号。

#### (2) 传输控制

- FRAME# 标志传输开始与结束。
- IRDY# Master 可以传输数据的标志。
- DEVSEL# 当 Slave 发现自己被寻址时设置低电平应答。
- TRDY# Slave 可以传输数据的标志。
- STOP# Slave 主动结束数据传输。
- IDSEL# 在即插即用系统启动时用于选中板卡的信号。

#### (3) 地址与数据总线

- AD[31:0] 地址/数据分时复用总线。
- C/BE# [3::0] 命令/字节势能信号。
- PAR 奇偶校验信号。



## (4) 仲裁信号

- REQ# Master 用来请求总线使用权。
- GNT# 仲裁机构允许 Master 得到总线使用权。

## (5) 错误报告

- PERR# 数据奇偶校验错。
- SERR# 系统奇偶校验错。

当 PCI 总线进行读操作时如图 2-50 所示。发起者先置 REQ#, 当得到仲裁器的许可时 (GNT#), 将 FRAME# 置低电平, 并在 AD 总线上放置 Slave 地址, 同时 C/BE# 放置命令信号, 说明接下来的类型传输。PCI 总线上的所有设备都需对此地址译码, 被选中的设备置 DEVSEL# 以声明自己被选中。然后当 IRDY# 与 TRDY# 都被置低时, 传输数据。Master 在数据传输结束前, 将 FRAME# 置高以表明只剩最后一组数据要传输, 并在传输完数据后, 放开 IRDY# 以释放总线控制权。由此可见, PCI 总线的传输是高效的, 发出一组地址后, 理想状态下可以连续发数据, 峰值速率为 132MB/s。

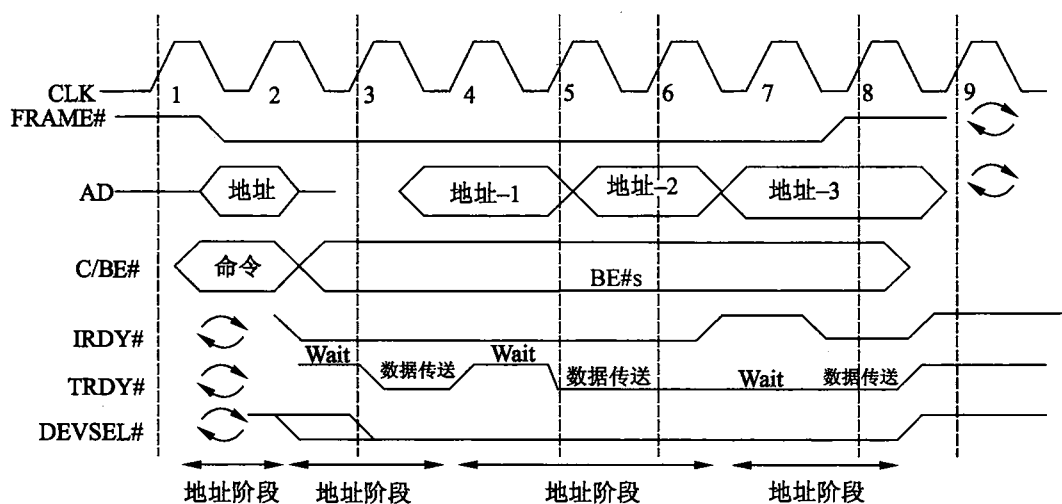


图 2-50 PCI 总线读操作

## 2.4.4 USB 接口基本原理与结构

USB (Universal Serial Bus, 通用串行总线) 是由 Intel 等厂商制定的连接计算机与具有 USB 接口的多种外设之间通信的串行总线。目前, 嵌入式系统中带 USB 接口的设备越来越多, 如鼠标、键盘、数码相机、Modem、扫描仪、摄像机、电视及视频抓取盒、音



箱等。

USB（通用串行总线）接口正在被用于多种嵌入式系统设备的数据通信中，如移动硬盘、数码相机、PDA、高速数据采集设备等。USB 通用串行总线是由 Compaq、HP、Intel、Lucent、Microsoft、NEC 和 Philips 这 7 家公司联合推出的新一代标准接口总线。该总线是一种连接外围设备的机外总线，最多可连接 127 个设备，为微机系统扩充和配置外部设备提供了方便。

#### 1) USB 总线的主要性能特点

- 使用简单。USB 提供机箱外的热即插即用功能，连接外设不必再打开机箱，也不必关闭主机电源，USB 可智能地识别 USB 链上外围设备的动态插入或拆除，具有自动配置和重新配置外设的能力，因此连接设备方便。
- 每个 USB 系统中有个主机，USB 总线采用“级联”方式可连接多个外部设备。每个 USB 设备用一个 USB 插头连接到上一个 USB 设备的 USB 插座上，而其本身又提供一或多个 USB 插座供下一个或多个 USB 设备连接使用。这种多重连接是通过集线器（Hub）来实现的，整个 USB 网络中最多可连接 127 个设备，支持多个设备同时操作。
- 应用范围广。USB 系统数据报文附加信息少，带宽利用率高，可同时支持同步传输和异步传输两种传输方式。USB 设备的带宽可从几 Kb/s 到几 Mb/s（在 USB2.0 版本，最高可达几百 Mb/s）。一个 USB 系统可同时支持不同速率的设备，USB 总线既可连接键盘、鼠标、摄像头、游戏设备、虚拟现实外设等低速设备，也可连接电话、声频、麦克风、压缩视频这样的全速设备，还可连接视频、存储器、图像等高速设备。此外，USB 还允许复合设备（即具有多种功能的外设）连接到 PC 机。
- 低成本的电缆和连接器。USB 通过一根四芯的电缆传送信号和电源，电缆长度可变，可长达 5m。USB 统一的 4 引脚插头将取代机箱后部众多的串行接口、并行接口、键盘接口等插头。
- 较强的纠错能力。USB 系统可实时地管理设备插拔。在 USB 协议中包含了传输错误管理、错误恢复等功能，同时根据不同的传输类型来处理传输错误。
- 较低的协议开销带来了高的总线性能，且适合于低成本外设的开发。
- 支持主机与设备之间的多数据流和多消息流传输，且支持同步和异步传输类型。
- 总线供电。USB 总线可为连接在其上的设备提供 5V 电压/100mA 电流的供电，最大可提供 500mA 的电流。USB 设备也可采用自供电方式。

#### 2) USB 系统描述

一个 USB 系统由 3 部分来描述：USB 主机、USB 设备和 USB 互连。

### (1) USB 设备

USB 设备分为 Hub (集线器) 和 Function (功能) 两大类。Hub 提供到 USB 的附加连接点, Function 为主机系统提供附加的性能, 如 ISDN 连接、数字操纵杆或扬声器等。实际上, 功能就是可发送和接收 USB 数据的、可实现某种功能的 USB 设备。USB 设备应具有标准的 USB 接口。

### (2) USB 主机

在任一 USB 系统中只有一个主机, 到主计算机系统的 USB 接口被称作主控制器。主控制器可采用硬件、固件或软件相结合的方式来实现。与 Hub 集成在主机系统内, 向上与主总线 (如 PCI 总线) 相连, 向下可提供一或多个连接点。

### (3) USB 互连

USB 互连指的是 USB 设备与主机的连接和通信方式, 它包括总线拓扑结构、内层关系、数据流模型和 USB 调度表。

USB 总线用来连接各 USB 设备和 USB 主机。USB 在物理上连接成一个层叠的星形拓扑结构, Hub 是每个星的中心, 每根线段表示一个点到点 (Point-to-Point) 的连接, 可以是主机与一个 Hub 或功能之间的连接, 也可以是一个 Hub 与另一个 Hub 或功能之间的连接。

由于对 Hub 和电缆传输时间的定时限制, USB 的拓扑结构最多只能有 7 层 (包括根层)。在主机和任一设备之间的通信路径中最多支持 5 个非根 Hub, 复合设备 (Compound Device) 要占据两层, 不能把它连到第 7 层, 第 7 层只能连接 Function 设备。

### 3) 物理接口

USB 总线的电缆有 4 根导线: 一对标准尺寸的双绞信号线和一对标准尺寸的电源线。

USB 总线支持的数据传输率有 3 种: 高速信令位传输率为 480Mbps; 全速信令位传输率为 12Mb/s; 低速信令位传输率为 1.5Mb/s。

USB2.0 支持在主控制器与 Hub 之间用高速传输全速和低速数据, 而 Hub 与设备之间以全速或低速传输数据, 这种支持能力可以将全速设备和低速设备对高速设备可用带宽的影响减到最小。

### 4) 电源

USB 的电源规范包括两个方面:

- 电源分配用来处理 USB 设备如何使用主机通过 USB 总线提供的电源。
- 电源管理用来处理 USB 系统软件和设备如何适应主机上的电源管理系统。

#### (1) 电源分配

每根 USB 电缆提供的电源功率是有限的, 主机为直接连接到它的 USB 设备提供电源, Hub 也对它所连接的 USB 设备提供电源, USB 设备也可自带电源。完全依赖电缆供电的

USB 设备称作总线供电设备 (Bus-Powered Device), 有后备 (Alternate) 电源的设备称作自我供电设备 (Self-Powered Device)。

## (2) 电源管理

USB 主机有一个独立于 USB 的电源管理系统, USB 系统软件与主机电源管理系统之间交互作用, 共同处理诸如挂起或恢复这样的系统电源事件。

## 5) 总线协议

USB 是一种查询 (Polling) 总线, 由主控制器启动所有的数据传输。USB 上所挂连的外设通过由主机调度的 (Host-Scheduled)、基于令牌的 (Token-Based) 协议来共享 USB 带宽。

大部分总线事务涉及 3 个包的传输。当主控制器按计划地发出一个描述事务类型和方向、USB 设备地址和端点号的 USB 包时, 就开始发起一个事务, 这个包称作“令牌包” (Token Packet), 它指示总线上要执行什么事务, 欲寻址的 USB 设备及数据传送方向。然后, 事务源发送一个数据包 (Data Packet), 或者指示它没有数据要传输。最后, 目标一般还要用一个指示传输是否有成功的握手包 (Handshake Packet) 来响应。

主控制器和 Hub 之间的某些总线事务涉及 4 个包的传输, 这些类型的事务用来管理主机与全/低速设备之间的数据传输。

主机与设备端点之间的 USB 数据传输模型被称作管道。管道有两种类型: 流和消息。消息数据具有 USB 定义的结构, 而流数据没有。管道与数据带宽、传输服务类型、端点特性 (如方向性和缓冲区大小) 有关。当 USB 设备被配置时, 大多数管道就形成了。一旦设备加电, 总是形成一个被称作默认控制管道的消息管道, 以便提供对设备配置、状态和控制信息的访问。

事务调度表 (Transaction Schedule) 允许对某些流管道进行流量控制, 在硬件级, 通过使用 NAK (否认) 握手信号来调节数据传输率, 以防止缓冲区上溢或下溢产生。当被否认时, 一旦总线时间可用会重试该总线事务。流量控制机制允许灵活地进行调度, 以适应异类混合流管道的同时服务, 因此, 可以在不同的时间间隔, 用不同规模的包为多个流管道服务。

## 6) 健壮性 (Robustness)

USB 采取以下措施提高它的健壮性:

- 使用差分驱动器和接收器以及屏蔽保护, 以保证信号的完整性。
- 控制域和数据域的 CRC 保护校验。
- 连接和断开检测及系统级资源配置。
- 协议的自我修复, 对丢失包或毁坏包执行超时 (Timeouts) 处理。

- 对流数据进行流量控制, 以保证对等步和硬件缓冲器维持正常的管理。
- 采用数据管道和控制管道结构, 以保证功能之间的独立性。

协议允许用硬件或软件的方法对错误进行处理, 硬件错误处理包括对传输错误的报告和重发。通知客户软件传输失败之前, USB 主控制器将会试着传送, 直到错误达到 3 次为止。

#### 7) USB 接口工作原理

USB 设备最大的特点就是即插即用, 之所以能够这样, 是因为 USB 协议规定在主机启动或是 USB 设备插入系统的时候都要对设备进行配置。这里所说的配置, 就是按照 USB 协议, 在 USB 主机与 USB 设备之间进行的一系列“问答”的过程。这一问答过程是通过主机与 USB 设备的端点 0 进行通信来完成的。USB 设备在插入 USB 端点时, 主机都通过默认地址 0 与设备的端点 0 进行通信。在这个过程中, 主机发出一系列试图得到描述符的标准请求, 通过这些请求, 主机得到所有感兴趣的设备信息, 从而知道了设备的情况以及该如何与设备通信。随后主机通过发出 Set Address 请求为设备设置一个唯一的地址。这样, 配置过程就完成了, 以后主机就通过为设备设置好的地址与设备通信, 而不再使用默认地址 0。有时设置完地址后, 可能还要获取一次描述符, 然后再设置配置 (Set Configuration) 之后才算完成了对新插入 USB 总线设备的配置过程。

在配置阶段主机也了解了设备端点的使用情况, 便可以通过这些端点来进行特定传输方式的通信。例如对于标准 USB 设备 U 盘来说, 可以采用批量传输方式 (BULK), 并使用特定的 BULK 端点。由于 U 盘是一个标准的 USB 设备, 操作系统带有它的驱动, 而不需要编写专门的主机驱动程序。但这样就必须为它选择一种标准命令集, 如 UFI 或 SCSI。这种情况下, 操作系统中自带的驱动就可以管理这个设备了。但对于非标准设备, 则可以自定义一套请求指令集, 同样采用 BULK 传输方式, 但需要编写专门的主机驱动程序来实现对 USB 设备的操作。因此, 在使用该设备之前必须安装设备驱动程序。

### 2.4.5 SPI 接口基本原理与结构

SPI (Serial Peripheral Interface, 串行外围设备接口) 是由 Motorola 公司开发, 用来在微控制器和外围设备芯片之间提供一个低成本、易使用的接口 (SPI 有时候也被称为 4 线接口)。这种接口可以用来连接存储器 (存储数据)、A/D 转换器、D/A 转换器、实时时钟日历、LCD 驱动器、传感器、音频芯片, 甚至其他处理器。支持 SPI 的元件很多, 并且还在增加。

与标准的串行接口 (将在第 10 章讲到) 不同, SPI 是一个同步协议接口, 所有的传输都参照一个共同的时钟, 这个同步时钟信号由主机 (处理器) 产生, 接收数据的外设 (从设备) 使用时钟来对串行比特流的接收进行同步化。可能会有许多芯片连到主机的同一个

SPI 接口上, 这时主机通过触发从设备的片选输入引脚来选择接收数据的从设备, 没有被选中的外设将不会参与 SPI 传输。

SPI 主要使用 4 个信号: 主机输出/从机输入 (MOSI)、主机输入/从机输出 (MISO)、串行 SCLK 或 SCK 和外设芯片 ( $\overline{CS}$ )。有些处理器有 SPI 接口专用的芯片选择, 称为从机选择 ( $\overline{SS}$ )。

MOSI 信号由主机产生, 从机接收。在有些芯片上, MOSI 只被简单的标为串行输入 (SI), 或者串行数据输入 (SDI)。MISO 信号由从机产生, 不过还是在主机的控制下产生的。在一些芯片上, MISO 有时被称为串行输出 (SO) 或串行数据输出 (SDO)。外设片选信号通常只是由主机的备用 I/O 引脚产生的。图 2-51 是微处理器通过 SPI 和外设进行连接的示意图。

主机和外设都包含一个串行移位寄存器, 主机通过向它的 SPI 串行寄存器写入一个字节来发起一次传输。寄存器是通过 MOSI 信号线将字节传送给外设, 外设也将自己移位寄存器中的内容通过 MISO 信号线返回给主机, 如图 2-52 所示。这样, 两个移位寄存器中的内容就被交换了。外设的写操作和读操作是同步完成的, 因此 SPI 成为一个很有效的协议。

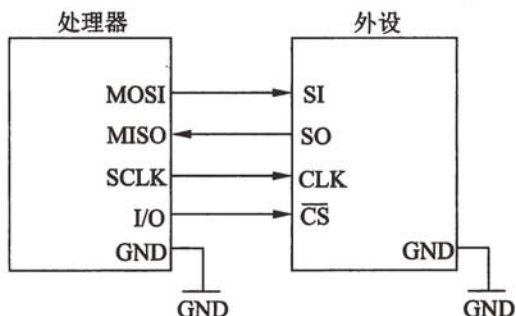


图 2-51 基本 SPI 接口

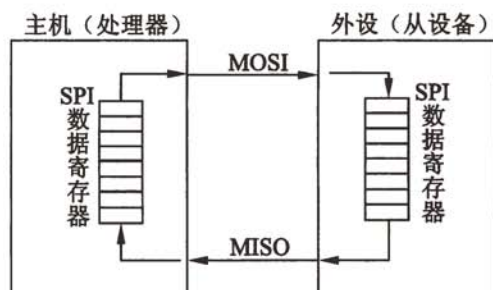


图 2-52 SPI 传输

如果只是进行写操作, 主机只需忽略收到的字节; 反过来, 如果主机要读取外设的一个字节, 就必须发送一个空字节来引发从机的传输。

当主机发送一个连续的数据流时, 有些外设能够进行多字节传输。许多拥有 SPI 接口的存储器芯片都以这种方式工作。在这种传输方式下, SPI 外设的芯片选择端必须在整个传输过程中保持低电平。比如, 存储器芯片会希望在一个“写”命令之后紧接着收到的是 4 个地址字节 (起始地址), 这样后面接收到的数据就可以存储到该地址。一次传输可能会涉及千字节的移位或更多的信息。

其他外设只需要一个单字节 (比如一个发给 A/D 转换器的命令), 有些甚至还支持菊花链连接, 如图 2-53 所示。

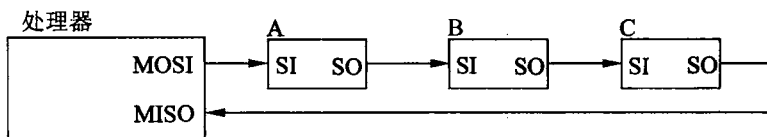


图 2-53 菊花链连接 3 台 SPI 设备

在这个例子中，主机处理器从其 SPI 接口发送 3 个字节的数据。第 1 个字节发送给外设 A，当第 2 个字节发送给外设 A 的时候，第 1 个字节已移出了 A，而传送给 B。同样，主机想要从外设 A 读取一个结果，它必须再发送一个 3 字节（空字节）的序列，这样就可以把 A 中的数据移到 B 中，然后再移到 C 中，最后送回到主机。在这个过程中，主机还依次从 B 和 C 接收到字节。

注意，菊花链连接不一定适用于所有的 SPI 设备，特别是要求多字节传输的（比如存储器芯片）设备。另外，要对外设芯片的数据表进行仔细分析，确定能对它做什么而不能做什么。如果芯片的数据表中没有明确提到菊花链连接，那么该芯片不支持这种连接的几率为 50%。

根据时钟极性和时钟相位的不同，SPI 有 4 个工作模式。时钟极性有高、低两极：时钟极性为低电平时，空闲时时钟（SCK）处于低电平，传输时跳转到高电平；时钟极性为高电平时，空闲时时钟处于高电平，传输时跳转到低电平。

时钟相位有两个：时钟相位 0 和时钟相位 1。对于时钟相位 0，如果时钟极性是低电平，MOSI 和 MISO 输出在（SCK）的上升沿有效。如果时钟电平极性为高，对于时钟相位 0，这些输出在 SCK 的下降沿有效。MISO 输出的第 X 位是一个未定义的附加位，是 SPI 接口特有的情况。用户不必担心这个位，因为 SPI 接口将忽略该位。

## 2.4.6 IIC 接口基本原理与结构

### 1. 概述

IIC BUS（Inter Integrated Circuit BUS，内部集成电路总线）是由 Philips 公司推出的二线制串行扩展总线，用于连接微控制器及其外围设备。IIC 总线产生于 20 世纪 80 年代，最初为音频和视频设备开发。IIC 总线是具备总线仲裁和高低速设备同步等功能的高性能多主机总线。由于其组成系统结构简单，无需专门的母板和插座，直接用导线连接设备，通信时无需片选信号，它支持任何一种 IC 制造工艺，且能够提升硬件的效率和简化电路的设计，因此众多厂商都提供了 IIC 兼容芯片；且 IIC（内部集成电路）总线是价格非常低、却很有效的用于互连小规模嵌入式系统内的外设网络。

在 IIC 总线上，只需要两条线——串行数据 SDA 线和串行时钟 SCL 线，它们用于总

线上器件之间的信息传递。每个器件都有一个唯一的地址以供识别，而且各器件都可以作为一个发送器或接收器（由器件的功能决定）。

IIC 总线有如下操作模式：主发送模式、主接收模式、从发送模式、从接收模式。下面介绍其通用传输过程及格式。

### （1）起始条件和停止条件

当 IIC 接口处于从模式时，要想数据传输，必须检测 SDA 线上的起始条件，起始条件由主器件产生。起始条件发生在 SCL 信号为高时，SDA 产生一个由高变低的电平变化处。当 IIC 总线上产生了一个起始条件，那么这条总线就被发出起始条件的主器件占用了，变成“忙”状态；停止条件发生在 SCL 信号为高时，SDA 产生一个由低变高的电平变化处。停止条件也由主器件产生，作用是停止与某个从器件之间的数据传输。当 IIC 总线上产生了一个停止条件，那么在几个时钟周期之后总线就被释放，变成“闲”状态。

当主器件送出一个起始条件，它还会立即送出一个从地址，来通知将与它进行数据通信的从器件。1 个字节的地址包括 7 位的地址信息和 1 位的传输方向指示位，如果第 7 位为 0，表示马上要进行一个写操作；如果为 1，表示马上要进行一个读操作。

### （2）数据传输格式

SDA 线上传输的每个字节长度都是 8 位，每次传输中字节的数量是没有限制的。在起始条件后面的第一个字节是地址域，之后每个传输的字节后面都有一个应答（ACK）位。传输中串行数据的 MSB（字节的高位）首先发送。

### （3）应答信号

为了完成 1 个字节的传输操作，接收器应该在接收完 1 个字节之后发送 ACK 位到发送器，告诉发送器，已经收到了这个字节。ACK 脉冲信号在 SCL 线上第 9 个时钟处发出（前面 8 个时钟完成 1 个字节的传输，SCL 上的时钟都是由主器件产生的）。当发送器要接收 ACK 脉冲时，应该释放 SDA 信号线，即将 SDA 置高。接收器在接收完前面 8 位数据后，将 SDA 拉低。发送器探测到 SDA 为低，就认为接收器成功接收了前面的 8 位数据。

## 2. IIC 总线工作原理

IIC 总路线用两线来连接多支路总线中的多个设备。这种总线是双向、低速的，并与公共时钟同步。可以直接将一个设备接到 IIC 总线上或是从该总线上取下，而不会影响其他设备。一些生产商，如 Microchip 公司、Philips 公司、Intel 公司等生产的小型微处理器都内置有 IIC 接口。IIC 总线的数据传输率比 SPI 总线要慢一些，在标准模式下传输速度为 100Kb/s，在快速模式下为 400Kb/s。

利用 IIC 接口在设备之间进行连接所使用的两根线是 SDA（串行数据）和 SCL（串行时钟），它们都是开漏（open-drain），通过一个上拉电阻接到正电源，因此在不使用的时候

仍保持高电平。使用 IIC 总线进行通信的设备驱动这两根线变为低电平，在不使用时就让它们保持高电平。每个连到 IIC 的设备都有一个唯一地址，这个设备可以是数据发送者（总线主机）、接收者（总线从机），也可以二者都是。IIC 是多主机总线，这意味着可以有多个设备充当总路线主机的角色。

SDA 和 SCL 都是双向的。SPI 总线有两根单独的数据线，分别用于两个方向的通信，而 IIC 总线共享同一根信号线来完成主机传送和外设响应；另外，与 SPI 总线具有多个工作模式不同的是，IIC 总线只有一个工作模式。时钟、SCL 和数据线 SDA 之间的时序关系很简单直观：当空闲的时候，SDA 和 SCL 都是高电平，只有 SDA 变为低电平，接着 SCL 也变为低电平时才开始 IIC 总线的数据传输。

当 SDA 和 SCL 都变为低电平时，就是告诉总线上的所有接收设备：数据包的传输开始了。在 SCL 变为低电平后，SDA 才发送（高电平或者低电平）第一个有效数据位，这被称为开始条件。

对于被传输的每一位，当 SCL 为低电平时，在 SDA 上位必须变为有效。该位是在 SCL 的上升沿对 SDA 上的数据位进行采样的，也必须一直保持有效直到 SCL 再次变为低电平，然后 SDA 就在 SCL 再次变为高电平之前传输下一个位。

最后，SCL 变回高电平（无效），接着 SDA 也变为高电平，数据传输结束，这被称为结束条件。

无论多大的数据包都可以通过 IIC 总线进行传输。像 SPI 总线一样，IIC 也是高位先传输。如果数据接收者无法再接收更多的数据，它可以通过将 SCL 保持低电平来中断传输。这样可以迫使数据发送者等待，直到 SCL 被重新释放。

IIC 总线在传送数据过程中共有 3 种类型信号，它们分别是开始信号、结束信号和应答信号。

- 开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。
- 结束信号：SCL 为低电平时，SDA 由低电平向高电平跳变，结束传送数据。
- 应答信号：接收数据的 IC 在接收到 8 位数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

发送方发出的每个字节都必须经过接收方确认，每个字节的第 8 个数据位一旦传送结束，发送方就释放数据线 SDA。然后主机在 SCL 上产生一个额外的时钟脉冲，这会触发接收方通过将 SDA 置为低电平来表示对接收到的字节进行；如果接收方没能将 SDA 置为低电平，发送方就会中断传输，并且采取适当的错误处理措施。

IIC 是多主机总线，因此存在同一时间会有多个主机试图开始数据传输的可能。由于



IIC 总线默认的状态是高电平，所以一个主机发送一个数据位 0 时会将 SDA 置为低电平，而如果这个数据位是 1 时就将总线置为默认状态。所以，如果两个主机要同时传输数据，一个主机发送数据位 1 将总线置为默认状态，但又检测到总线被另外一个主机置为了低电平（发送数据位 0），该主机将记录一个错误状态，并且终止数据的传输。

### 2.4.7 PCMCIA 接口基本原理与结构

PCMCIA (Personal Computer Memory Card International Association, PC 机内存卡国际联合会) 是一个成立于 1989 年的国际性组织，现在已经拥有 2000 多个企业会员，其中有 Intel、AMD、IBM、Compaq 和 TI 等国际知名公司。该组织成立指出，是为了建立一个物理尺寸较小、低功耗的、灵活的存储卡标准，以满足笔记本电脑对移动存储方面越来越迫切的要求。

1990 年 9 月，PCMCIA 推出了 PCMCIA 1.0 规范，该规范是针对各类存储卡或虚拟盘设计的，其接口采用了 JEIDA (Japanese Electronics Industry Development Association) 68 引脚的接口；1991 年，PCMCIA 推出了 2.0 规范，根据业界的需要添加了对 I/O 设备的规范，以方便在笔记本电脑上扩展 I/O 设备，但接口仍然沿用了 1.0 规范兼容的 68 引脚的接口；同时，PCMCIA 对其驱动程序的架构也作了规范，以便于软件开发人员开发的驱动程序可以相互兼容。

随着多媒体和高速网络的发展，PCMCIA 原有的 16 位体系结构无法满足快速的要求，于是又开发了 32 位的 CardBus。到现在，基于 PCMCIA 的设备已经相当成熟，不仅在笔记本电脑上得到了广泛的应用，在很多电子产品，如数码相机、机顶盒、车载设备、手持设备、PDA 等方面也不断地被采用。现在越来越多的产品都需要接口具有可扩展模块化的功能，因此 PCMCIA 也将自己的使命改成了“发展模块化外设的标准，并将他们推广到全世界”。

物理上 PCMCIA 被定义成 68 引脚的接口，但是，卡片有 16 位和 32 位之分。16 位的 PCMCIA 卡通常叫 PCCard，它的速度较慢，其时序和 ISA 总线类似。32 位的 PCMCIA 标准称做 CardBus，其运行频率达到 33MHz，速度已经能够满足一般局域网及宽带应用的要求。现在只在桌面和较大系统上才拥有的高级功能可以移入 CardBus 卡，从而可以用在移动环境下。CardBus 接口的信号传输协议起源于 PCI 局部总线信号传输协议，它支持以任何组合形式实现多个总线功能。总线主控功能可为处理器分担任务，有利于在多任务环境中改善系统的吞吐量。

然而，CardBus 结构相对复杂，在现在的很多嵌入式处理器的应用中，对 CardBus 的支持需要专门的总线（如 PCI）或外设等，而对 PCCard 支持则要容易得多，因为时序和 ISA 类似，从嵌入式处理器的 Host Bus 接口扩展对 PCCard 的支持，需要的逻辑也很简单；

而且,同时也可以提供对 CF 卡的支持。因此这里主要讲述 16 位 PCMCIA 接口的规范与结构。

PCMCIA 卡的外形尺寸有 5 种,分别为 I 型(TYPE I)、II 型(TYPE II)和 III 型(TYPE III)、扩展 TYPE I 和扩展 TYPE II,其中 I 型~III 型的 PCMCIA 卡的长度均为 85.60mm×54.00mm,而卡的厚度分别为 3.3mm、5.0mm 和 10.5mm;而扩展 TYPE I 和扩展 TYPE II 的 PCMCIA 卡主要是为了兼容某些尺寸较大的接口,如 RJ45 接口等。

PCMCIA 卡最初设计的供电电压是 5V,在其 2.0 和 2.1 规范中增加了对 3.3V 电压的支持,但为了兼容以前供电电压为 5V 的卡,在卡插入插槽的时候,仍然是 5V 供电;当系统识别到该卡,确认其支持的电压后再以 3.3V 供电。在新的标准中,可以将 PCMCIA 卡直接以 3.3V 开始工作,因此 PCMCIA 卡的插槽在这一点上也做了改动,以免这些工作在低电压的插卡不会错误地插入原来 5V 的插槽中。

为了让新的插槽能够兼容以前 5V 的插卡,PCMCIA 规范中添加了两个电压敏感 VS (Voltage Sense) 信号识别插入的 PCMCIA 卡的工作电压。

一般而言 PCMCIA 接口和系统总线接口需要一个 HBA (Host Bus Adapter) 运行转换。这个 HBA 可以是一个芯片,也可以是一些逻辑。

PCMCIA 内存卡规范:PCMCIA 规范里一共定义了 6 类 PCMCIA 卡,分别是:内存卡、I/O 卡(内存或 I/O)、硬盘 ATA (AT Attachment for IDE drivers) 接口、DMA (Direct Memory Access) 接口、AIMS (Auto-Indexing Mass Storage) 和 32 位 PC 卡接口 CardBus。

## 2.5 嵌入式系统网络接口

### 2.5.1 以太网接口基本原理与结构

以太网(Ethernet)和 TCP/IP 协议已经成为使用最广泛的通信协议。它的高速、可靠、分层及可扩充性使得它在各个领域的应用越来越灵活。

#### 1. 以太网基础知识

在这里将以 10MB 的以太网协议为例,说明以太网传输的物理层和 Mac 层的协议。

最常用的以太网协议是 IEEE802.3 标准。嵌入式系统经常使用该标准,下面对其进行介绍。

##### (1) 传输编码

在 802.3 版本的标准中,没有采用直接的二进制编码(即用 0V 表示 0,用 5V 表示 1),因为这样做将产生歧义。如果某个以太网控制器发送了串行数据 0001000,接收方可能认为是 1000000 或者 0100000,因为它们不能区分无法送(0V)和比特 0(0V)。所以,就要

求在没有同步时钟的情况下,接收方能够明确地定位比特的开始和结束。曼彻斯特编码(Manchester Encoding)与差分曼彻斯特编码(Differential Manchester Encoding)就是实行这种功能的两种解决方案。

各种编码的时序如图 2-54 所示。

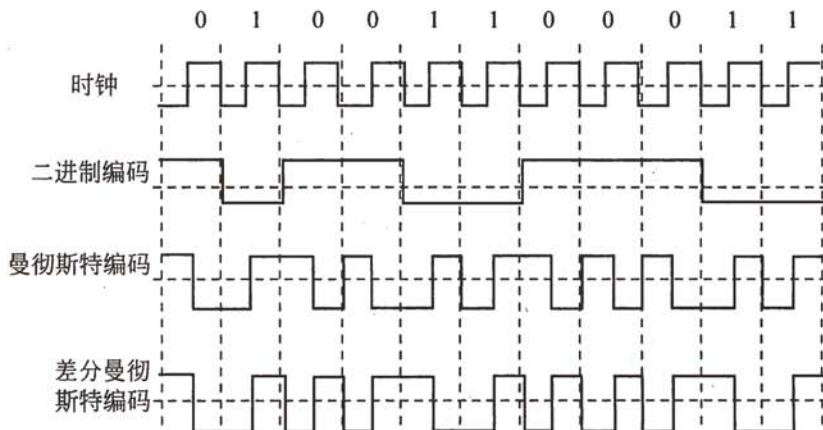


图 2-54 各种编码的时序

曼彻斯特编码的规律是:每位中间有一个电平跳变,从高到低的跳变表示为“0”,从低到高的跳变表示为“1”。

差分曼彻斯特编码的规律是:每位的中间也有一个电平跳变,但不用这个跳变来表示数据,而是利用每个码元开始时有无跳变来表示“0”或“1”,有跳变表示“0”,无跳变表示“1”。

曼彻斯特编码和差分曼彻斯特编码相比,前者编码简单,后者能提供更好的噪声抑制性能。在所有的 802.3 系统中,都采用曼彻斯特编码,其高电平为+0.85V,低电平信号为-0.85V,这样指令信号电压仍然是 0V。

## (2) 802.3Mac 层的帧

802.3 Mac 层的以太网的物理传输帧如表 2-16 所示。

表 2-16 802.3 帧的格式

PR	SD	DA	SA	TYPE	DATA	PAD	FCS
56位	8位	48位	48位	16位	不超过1500字节	可选	32位

- PR: 同步位,用于收发双方的时钟同步,同时也指明了传输的速率(10MB 和 100MB 的时钟频率不一样,所以 100MB 网卡可以兼容 10MB 网卡),是 56 位的

二进制数 101010101010...

- SD: 分隔位, 表示下面跟着的是真正的数据而不是同步时钟, 为 8 位的 10101011, 和同步位不同的是最后 2 位是 11 而不是 10。
- DA: 目的地址, 以太网的地址为 48 位 (6 个字节) 二进制地址, 表明该帧传输给哪个网卡。如果为 FFFFFFFF, 则是广播地址。广播地址的数据可以被任何网卡接收到。
- SA: 源地址, 48 位, 表明该帧的数据是哪个网卡发的, 即发送端的网卡地址, 同样是 6 个字节。
- TYPE: 类型字段, 表明该帧的数据是什么类型的数据, 不同协议的类型字段不同。如: 0800H 表示数据为 IP 包, 0806H 表示数据为 ARP 包, 814CH 是 SNMP 包, 8137H 为 IPX/SPX 包。小于 0600H 的值是用于 IEEE802 的, 表示数据包的长度。
- DATA: 数据段, 该段数据不能超过 1500B。因为以太网规定整个传输包的最大长度不能超过 1514B (14B 为 DA, SA, TYPE)。
- PAD: 填充位。由于以太网帧传输的数据包最小不能小于 60B, 除去 (DA、SA、TYPE 的 14B), 还必须传输 46B 的数据, 当数据段的数据不足 46B 时, 后面补 0 (通常是 0, 也可以补其他值)。
- FCS: 32 位数据校验位。32 位的 CRC 校验, 该校验由网卡自动计算, 自动生成, 自动校验, 自动在数据段后面填入。不需要软件管理的。
- 通常, PR、SD、PAD、FCS 这几个数据段都是网卡 (包括物理层和 Mac 层的处理) 自动产生的, 剩下的 DA、SA、TYPE、DATA 这 4 个段的内容是上层的软件控制的。

### (3) 以太网数据传输的特点

- 所有数据位的传输由低位开始, 传输的位流是用曼彻斯特编码。
- 以太网是基于冲突检测的总线复用方法, 由于篇幅所限, 冲突退避算法这里就不介绍了, 它是由硬件自动执行的。
- 以太网传输的数据段的长度, DA+SA+TYPE+DATA+PAD 最小为 60B, 最大为 1514B。
- 通常的以太网卡可以接收 3 种地址的数据, 一个是广播地址, 一个是多播地址 (或者叫组播地址, 在嵌入式系统中很少用到), 一个是它自己的地址。但有时, 用于网络分析和监控, 网卡也可以设置为接收任何数据包。
- 任何两个网卡的物理地址都是不一样的, 是世界上唯一的, 网卡地址由专门机构分配。不同厂家使用不同地址段, 同一厂家的任何两个网卡的地址也是唯一的。根据网卡的地址段 (网卡地址的前 3 个字节) 可以知道网卡的生产厂家。

## 2. 嵌入式以太网接口的实现

在嵌入式系统中增加以太网接口，通常有如下两种方法实现：

(1) 嵌入式处理器+网卡芯片（比如：RTL8019AS，CS8900 等）

这种方法对嵌入式处理器没有特殊要求，只要把以太网芯片连接到嵌入式处理器的总线上即可。此方法通用性强，不受处理器的限制，但是，处理器和网络数据交换通过外部总线（通常是并行总线）交换数据，速度慢，可靠性不高，电路板走线复杂。

(2) 带有以太网接口的嵌入式处理器

这种方法要求嵌入式处理器有通用的网络接口（比如：MII 接口）。通常这种处理器是面向网络应用而设计的。处理器和网络数据交换通过内部总线，速度快。

## 3. 在嵌入式系统中主要处理的以太网协议

TCP/IP 是一个分层的协议。每一层实现一个明确的功能，对应一个或者几个传输协议。每层相对于它的下层都作为一个独立的数据包来实现。典型的分层和每层上的协议如表 2-17 所示。

表 2-17 TCP/IP 协议的典型分层和协议

分 层	每层上的协议	分 层	每层上的协议
应用层 (Application)	BSD 套接字 (BSD Sockets)	数据链路层 (Data Link)	IEEE802.3 Ethernet MAC
传输层 (Transport)	TCP、UDP	物理层 (Physical)	
网络层 (Network)	IP、ARP、ICMP、IGMP		

(1) ARP (Address Resolution Protocol, 地址解析协议)

网络层用 32 位的地址来标识不同的主机（这就是人们熟知的 IP 地址），而链路层使用 48 位的物理（MAC）地址来标识不同的以太网或令牌环网接口。只知道目的主机的 IP 地址并不能发送数据帧给它，必须知道目的主机网络接口的物理地址才能发送数据帧。

ARP 的功能就是实现从 IP 地址到对应物理地址的转换。源主机发送一份包含目的主机 IP 地址的 ARP 请求数据帧给网上的每个主机，称作 ARP 广播，目的主机的 ARP 收到这份广播报文后，识别出这是发送端在询问它的 IP 地址，于是发送一个包含目的主机 IP 地址及对应的物理地址的 ARP 回答给源主机。

为了加快 ARP 协议解析的数据，每台主机上都有一个 ARP cache，存放最近的 IP 地址到硬件地址之间的映射记录。其中每一项的生存时间（一般为 20 分钟），这样当在 ARP 的生存时间之内连续进行 ARP 解析的时候，不需要反复发送 ARP 请求了。

(2) ICMP (Internet Control Messages Protocol, 网络控制报文协议)

ICMP 是 IP 层的附属协议，IP 层用它来与其他主机或路由器交换错误报文和其他重要控制信息。ICMP 报文是在 IP 数据包内部被传输的。在 Linux 或者 Windows 中，两个常用

的网络诊断工具 ping 和 traceroute (Windows 下是 Tracert), 其实就是 ICMP 协议。

### (3) IP (Internet Protocol, 网际协议)

IP 工作在网络层, 是 TCP/IP 协议族中最为核心的协议。所有的 TCP、UDP、ICMP 及 IGMP 数据都以 IP 数据包格式传输 (IP 封装在 IP 数据包中)。IP 数据包最长可达 65535 字节, 其中报头占 32 位。还包含各 32 位的源 IP 地址和 32 位的目的 IP 地址。

TTL (time-to-live, 生存时间字段) 指定了 IP 数据包的生存时间 (数据包可以经过的最多路由器数)。TTL 的初始值由源主机设置, 一旦经过一个处理它的路由器, 它的值就减去 1。当该字段的值为 0 时, 数据包就被丢弃, 并发送 ICMP 报文通知源主机重发。

IP 提供不可靠、无连接的数据包传送服务, 高效、灵活。

不可靠 (unreliable) 的意思是它不能保证 IP 数据包能成功地到达目的地。如果发生某种错误, IP 有一个简单的错误处理算法: 丢弃该数据包, 然后发送 ICMP 消息报给信源端。任何要求的可靠性必须由上层来提供 (如 TCP)。

无连接 (connectionless) 的意思是 IP 并不维护任何关于后续数据包的状态信息。每个数据包的处理是相互独立的。IP 数据包可以不按发送顺序接收。如果一信源向相同的信宿发送两个连续的数据包 (先是 A, 然后是 B), 每个数据包都是独立地进行路由选择, 可能选择不同的路线, 因此 B 可能在 A 到达之前先到达。

IP 的路由选择: 源主机 IP 接收本地 TCP、UDP、ICMP、IGMP 的数据, 生成 IP 数据包, 如果目的主机与源主机在同一个共享网络上, 那么 IP 数据包就直接送到目的主机上。否则就把数据包发往一默认的路由器上, 由路由器来转发该数据包。最终经过数次转发到达目的主机。IP 路由选择是逐跳 (hop-by-hop) 进行的。所有的 IP 路由选择只为数据包传输提供下一站路由器的 IP 地址。

### (4) TCP (Transfer Control Protocol, 传输控制协议)

TCP 协议是一个面向连接的可靠的传输层协议。TCP 为两台主机提供高可靠性的端到端数据通信。它所做的工作包括:

① 发送方把应用程序交给它的数据分成合适的小块, 并添加附加信息 (TCP 头), 包括顺序号, 源、目的端口, 控制、纠错信息等字段, 称为 TCP 数据包。并将 TCP 数据包交给下面的网络层处理。

② 接受方确认接收到的 TCP 数据包, 重组并将数据送往高层。

### (5) UDP (User Datagram Protocol, 用户数据包协议)

UDP 协议是一种无连接不可靠的传输层协议。它只是把应用程序传来的数据加上 UDP 头 (包括端口号, 段长等字段), 作为 UDP 数据包发送出去, 但是并不保证它们能到达目的地。可靠性由应用层来提供。就像发送一封写有地址的一般信件, 却不保证它能到达。

因为协议开销少, 和 TCP 协议相比, UDP 更适用于应用在低端的嵌入式领域中。很

多场合如网络管理 SNMP, 域名解析 DNS, 简单文件传输协议 TFTP, 大都使用 UDP 协议。

端口: TCP 和 UDP 采用 16 位的端口号来识别上层的 TCP 用户, 即上层应用协议, 如 FTP 和 TELNET 等。常见的 TCP/IP 服务都用众所周知的 1~255 之间的端口号。例如 FTP 服务的 TCP 端口号都是 21, Telnet 服务的 TCP 端口号都是 23。TFTP (简单文件传输协议) 服务的 UDP 端口号都是 69。256~1023 之间的端口号通常都是提供一些特定的 UNIX 服务。TCP/IP 临时端口分配 1024~5000 之间的端口号。

#### 4. 网络编程接口

BSD 套接字 (BSD Sockets) 使用的最广泛的网络程序编程方法, 主要用于应用程序的编写, 用于网络上主机与主机之间的相互通信。

很多操作系统都支持 BSD 套接字编程。例如, UNIX、Linux、VxWorks, Windows 的 Winsock 基本上是来自 BSD Sockets。

套接字 (Sockets) 分为 Stream Sockets 和 Data Sockets。Stream Sockets 是可靠性的双向数据传输, 对应使用 TCP 协议传输数据; Data Sockets 是不可靠连接, 对应使用 UDP 协议传输数。

下面给出一个使用套接字接口的 UDP 通信的流程。

UDP 服务器端和一个 UDP 客户端通信的程序过程:

(1) 创建一个 Socket:

```
sFd=socket(AF_INET, SOCK_DGRAM, 0)
```

(2) 把 Socket 和本机的 IP, UDP 口绑定:

```
bind(sFd, (struct sockaddr *)&serverAddr, sockAddrSize)
```

(3) 循环等待, 接收 (recvfrom) 或者发送 (sendfrom) 信息。

(4) 关闭 Socket, 通信终止:

```
close(sFd)
```

## 2.5.2 CAN 总线接口的基本原理与结构

### 1. CAN 总线概述

CAN (Controller Area Network, 控制器局域网) 总线是国际上应用最广泛的现场总线之一。最初, CAN 总线被汽车环境中的微控制器通信, 在车载各电子控制装置 (Electric Control Unit, ECU) 之间交换信息, 形成汽车电子控制网络。比如: 发动机管理系统、变速箱控制器、仪表装备、电子主干系统中均嵌入 CAN 总线控制装置。

一个由 CAN 总线构成的单一网络中, 理想情况下可以挂接任意多个节点, 实际应用中节点数目受网络硬件的电气特性所限制。例如: 当使用 Philips P82C250 作为 CAN 收发

器时,同一网络中允许挂接 110 个节点。CAN 可提供 1Mb/s 的数据传输速率,虽然相对于以太网并不算高速,但是,这足以使实时控制变得非常容易。而且,CAN 总线是一种多主方式的串行通信总线。基本设计规范要求有高的位速率,高抗电磁干扰性,并可以检测出产生的任何错误。当信号传输距离达到 10Km 时 CAN 总线仍可提供高达 50Kb/s 的数据传输速率。由于 CAN 总线具有很高的实时性能。它已经在汽车工业、航空工业、工业控制、安全防护等领域中得到了广泛应用。

CAN 总线能够使用多种物理介质进行传输,例如:双绞线、光纤等。最常用的就是双绞线。总线信号使用差分电压传送,两条信号线被称为 CAN\_H 和 CAN\_L,静态时均是 2.5V 左右,此时状态表示为逻辑 1 也可以叫做“隐性”。用 CAN\_H 比 CAN\_L 高表示逻辑 0,称为“显性”。此时,通常电压值为 CAN\_H=3.5V 和 CAN\_L=1.5V。当“显性”位和“隐性”位同时发送的时候,最后总线数值将为“显性”。正是这种特性,为 CAN 总线的仲裁奠定了基础。

CAN 总线的位时间可以分成四个部分:同步段,传播时间段,相位缓冲段 1 和相位缓冲段 2。每段的时间份额的数目都是可以通过 CAN 总线控制器编程控制,而时间份额的大小  $t_q$  由系统时钟  $t_{sys}$  和波特率预分频值 BRP 决定:  $t_q = BRP / t_{sys}$ 。图 2-55 说明了 CAN 总线的位时间的各个组成部分。

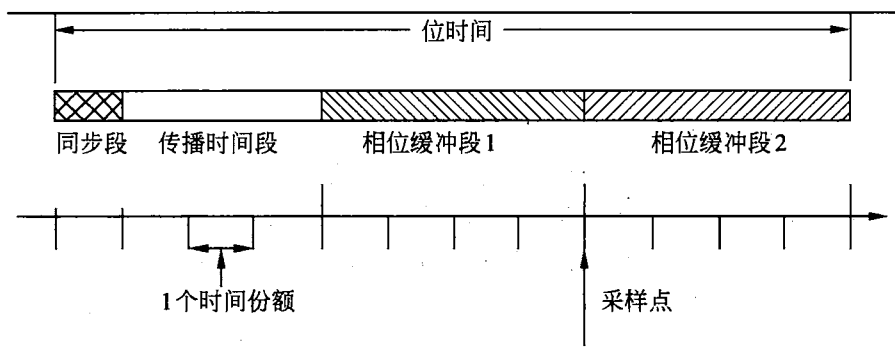


图 2-55 CAN 总线的位时间

- 同步段: 用于同步总线上的各个节点, 在此段内期望有一个跳变沿出现 (其长度固定)。如果跳变沿出现在同步段之外, 那么沿与同步段之间的长度叫做沿相位误差。采样点位于相位缓冲段 1 的末尾和相位缓冲段 2 开始处。
- 传播时间段: 用于补偿总线上信号传播时间和电子控制设备内部的延迟时间。因此, 要实现与位流发送节点的同步, 接收节点必须移相。CAN 总线非破坏性仲裁



规定, 发送位流的总线节点必须能够收到同步于位流的 CAN 总线节点发送的显性位。

- 相位缓冲段 1: 重同步时可以暂时延长。
- 相位缓冲段 2: 重同步时可以暂时缩短。
- 同步跳转宽度: 长度小于相位缓冲段。

上述几个部分的设定和 CAN 总线的同步、仲裁等信息有关。其主要思想是要求各个节点在一定误差范围内保持同步。必须考虑各个节点时钟(振荡器)的误差和总线的长度带来的延迟(通常每米延迟为 5.5ns)。正确设置 CAN 总线各个时间段, 是保证 CAN 总线良好工作的关键, 请参考 CAN 总线方面的相关资料。

按照 CAN2.0B 协议规定, CAN 总线的帧数据有如图 2-56 所示的两种格式: 标准格式和扩展格式。作为一个通用的嵌入式 CAN 节点, 应该支持上述两种格式。



图 2-56 CAN 总线数据帧格式

## 2. 在嵌入式处理器上扩展 CAN 总线接口

因为 CAN 总线是通用的现场总线标准, 对于一些面向工业控制的处理器, 本身就集成了一个或者多个 CAN 总线控制器。比如: 韩国现代公司的 hms30c7202 (ARM720T 内核) 带有两个 CAN 总线控制器, 而 Phillips 公司的 LPC2194 和 LPC2294 (ARM7TDMI 内核) 带有 4 个 CAN 总线控制器。CAN 总线控制器主要是完成时序逻辑转换等工作, 要在电气特性上满足 CAN 总线标准, 还需要一个转换芯片。用它来实现 TTL 电平到 CAN 总线电平特性的转换, 这就是 CAN 收发器, 通常称之为 CAN 总线的物理层芯片。

实际上, 多数嵌入式处理器都不带 CAN 总线控制器。在嵌入式处理器的外部总线上扩展 CAN 总线接口芯片是通用的解决方案。常用的 CAN 总线接口芯片主要有: Phillips 公司的 SJA1000 和 MicroChip 公司的 MCP251x 系列 (MCP2510 和 MCP2515)。这两种芯片都支持 CAN2.0B 标准。

SJA1000 是“51 时代”的产物, 其总线采用的是地址线 and 数据线复用的方式, 在 Intel 8051 兼容的总线上可以很方便的扩展, 而不需要单独的锁存器。但是, 当前嵌入式处理器

外部总线大多是地址线 and 数据线分开（而不是复用）的结构，使用 SJA1000 作为扩展，通常采用如图 2-57 所示结构。

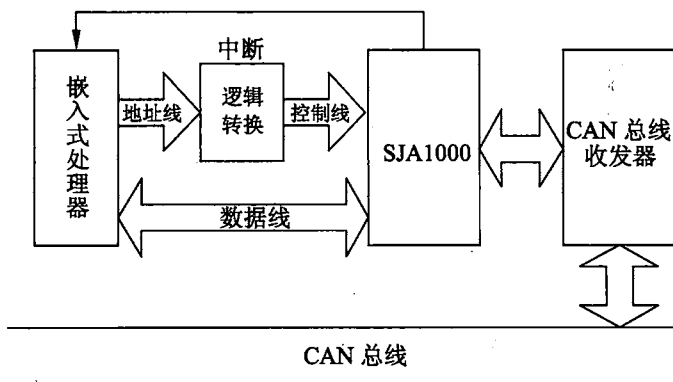


图 2-57 通过逻辑转换实现的嵌入式 CAN 节点

时序转换模式，通过一些逻辑，可以把从嵌入式处理器引出的总线信号转换成 SJA1000 的总线接口。每次对 SJA1000 操作的时候，需要先后写入地址和数据两次数据。

上述使用 SJA1000 扩展 CAN 总线的方法接口复杂。而且，主流的嵌入式处理器为了降低系统功耗，很少采用 5V 逻辑，而 SJA1000 使用的是 5V 逻辑。在系统设计的时候，还需要考虑电平兼容的问题。结果导致，系统功耗和成本的提高，可靠性的降低。

MicroChip 公司的 MCP2510（MCP2515）CAN 总线控制器的特点如下：

- 支持标准格式和扩展格式的 CAN 数据帧结构（CAN2.0B）
- 0~8 字节的有效数据长度，支持远程帧
- 最大 1Mb/s 的可编程波特率
- 两个支持过滤器（Filter、Mask）的接收缓冲区，三个发送缓冲区
- 支持回环（Loop Back）模式，便于测试
- SPI 高速串行总线，最大 5MHz
- 3V 到 5.5V 供电

近年来 SOC 技术迅速发展，大多数嵌入式处理器都有 SPI 总线控制器，如图 2-58 所示，可以直接和 MCP251x 连接。

MCP2510 可以 3V 到 5.5V 供电，因此能够直接和 3.3V I/O 口的嵌入式处理器连接。系统结构简单，与处理器之间的 SPI 串行接口，减少了总线的物理连接，提高了系统的可靠性。

这里，使用 MCP2510 在三星公司的 S3C44B0X 处理器上扩展 CAN 总线接口。其原理如图 2-59 所示。

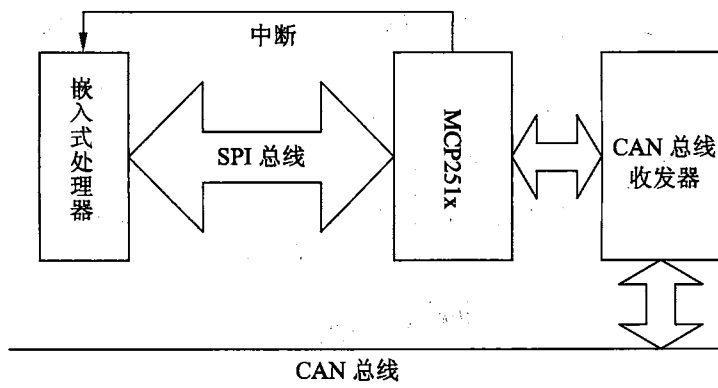


图 2-58 MCP251x 组成的嵌入式 CAN 节点

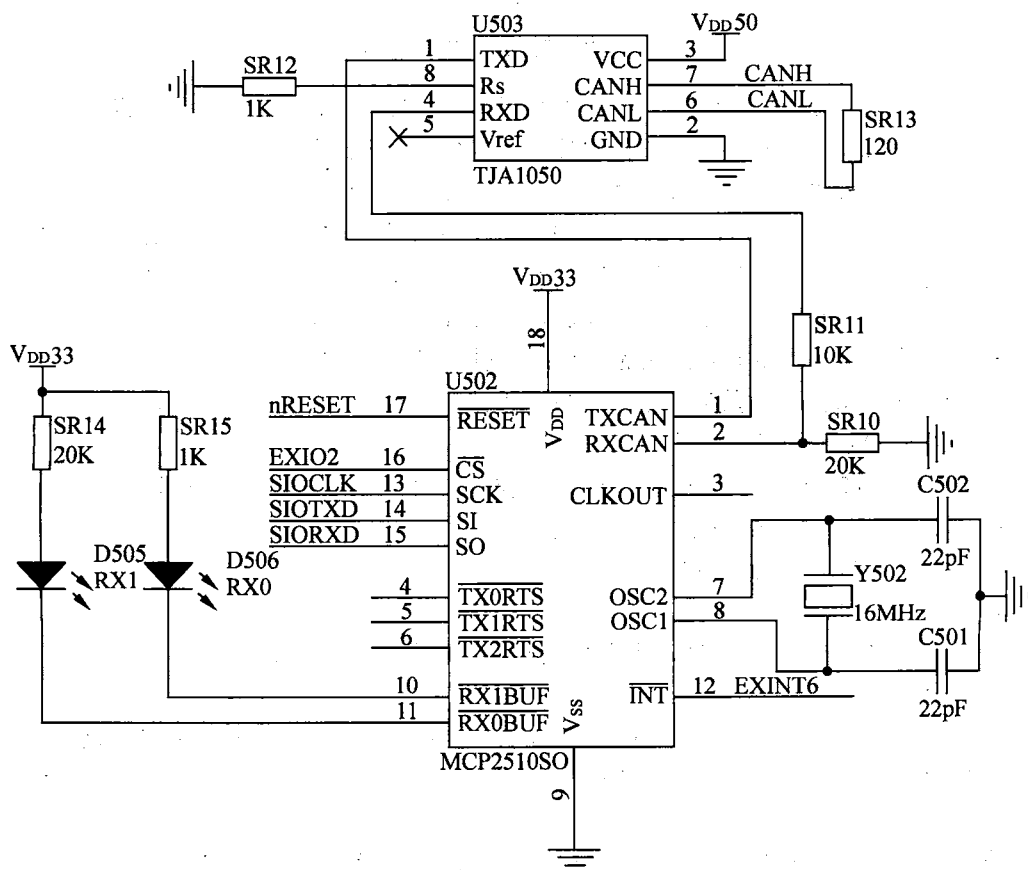


图 2-59 MCP2510 组成的 CAN 总线接口

在这个电路中, MCP2510 使用 3.3V 电压供电。它可以直接和 S3C44B0X 通过 SPI 总线(在 S3C44B0X 的 datasheet 中, 把这个接口叫 SIO, 同步串口)连接。相关的资源如下:

- 使用一个扩展的 I/O 口 (EXIO2) 作为片选信号, 低电平有效。
- 用 S3C44B0X 的外部中断 6 (EXINT6) 作为中断管脚, 低电平有效。
- 16MHz 晶体作为输入时钟, MCP2510 内部有振荡电路, 用晶体可以直接起振。
- 使用 TJA1050 作为 CAN 总线收发器。

但是, 这里存在一个问题, TJA1050 必须使用 5V 供电, 这就带来了电平兼容的麻烦。幸运的是, MCP2510 和 TJA1050 连接的两个信号都是单向的信号。对于 MCP2510, TXCAN 是输出信号, RXCAN 是输入信号。只需要单向满足 I/O 口的电气特性就可以了, 即:

- 从 MCP2510 输出的 TXCAN 信号, 是不是可以满足 TJA1050 的输入电平。TJA1050 为 5V 供电时, 输入高电平  $V_{ih}$  的范围是 2~5.3V。而 3.3V 供电的 MCP2510 输出的 TXCAN 信号高电平  $V_{oh}$  最小值为 2.6V, 可以满足要求。
- 从 TJA1050 输出的 RXD 信号, 是不是可以满足 MCP2510 的输入电平。当 MCP2510 用 3.3V 供电时, 输入信号 RXCAN 高电平的范围  $V_{ih}$  是 2~4.3V。这并不能满足 5V 逻辑的 TJA1050 输出电平, 因此需要一点转换工作。这里用电阻分压的方法实现单向的电平转换。这是一种简单有效的方法, 分压电阻值的选择, 需要考虑两个问题: ①TJA1050 输出信号的驱动能力; ②MCP2510 的 RXCAN 管脚的输入阻抗。前者, 最多可以输出 15mA 的电流, 这足以驱动电阻  $RS_{11}+RS_{10}=10K\Omega+20K\Omega=30K\Omega$ ; 而后者, 输入电流  $I_{LI}$  不会超过 5 $\mu$ A。

### 2.5.3 xDSL 接口基本原理与结构

xDSL (x Digital Subscriber Loop, 数字用户线路) 属于铜线传输技术, 它是美国贝尔通信研究所于 1989 年为推动 VOD (Video-On-Demand, 视频点播) 业务而开发出的以铜质电话线为传输介质的高速传输技术, 后因 VOD 业务受挫而被搁置了很长一段时间。近年来随着 Internet 技术, 特别是基于 Web 浏览业务的迅速发展, 人们对宽带业务的需求迅速增长, 使 xDSL 技术在宽带 Internet 接入领域获得了新的应用。

众所周知, 常用的公用电话网 (Public Switched Telephone Network, PSTN) 在交换机内采用了全数字化的交换技术, 但最终到达用户端的信号仍是模拟信号。按照这种方式将用户线接入全数字化的 Internet, 必须先用 Modem 将计算机的数字信号变换为模拟信号, 交换机则又必须将模拟信号转变为数字信号。经过 D/A 和 A/D 两次转换, 传输速率就会受到很大限制。xDSL (数字用户线路) 技术是在现有用户电话线两侧同时接入专用的 DSL

调制解调设备, 在用户线上利用数字信号高频带宽较宽的特性直接采用数字信号传输, 省去中间的 A/D 转换, 突破了模拟信号传输极限速率为 56KB/s 的限制。xDSL 技术为用户提供数 Mb/s 的数据传输速率, 是 Modem 数据传输速率的几十倍, 较好地满足了用户的需求。

### 1. xDSL 技术的分类

xDSL 中, x 代表着不同种类的数字用户线路技术。各种数字用户线路技术的不同之处, 主要表现在信号的传输速率和距离, 还有对称和非对称的区别上。

DSL 技术主要分为对称和非对称两大类。

#### (1) 对称 DSL 技术

- 高速 (DSL High-bit-rate DSL, HDSL)。
- 单线 DSL (Single-line DSL, SLDSL), 这是 HDSL 的单线版本, 它可以提供双向高速可变比特率连接, 速率范围为 160Kb/s~2.084Mb/s。
- MVL (Multiple Virtual Line, 多虚拟数字用户线), MVL 是 Paradyne 公司开发的低成本 DSL 传输技术。
- ISDN (Integrated Services Digital Network, 综合服务数字网) 数字用户线 (IDSL) 通过在用户端使用 ISDN 终端适配器和在双绞线的另一端使用与 ISDN 兼容的接口卡, 这种技术可以提供 128Kb/s 的服务。

#### (2) 非对称 DSL 技术

- 非对称 DSL (Asymmetric DSL, ADSL)。
- 速率自适应 DSL (Rate Adaptive DSL, RADSL), 这种技术允许服务提供者调整 xDSL 连接的带宽以适应实际需要并且解决线长和质量问题。
- 高速数字用户线 (Very High Data Rate DSL, VDSL)。

而其中主要的技术有 HDSL、ADSL、VDSL 3 种。此外还有一些公司所提出的建议, 如 UDSL (Unidirectional DSL)、x2/DSL 等, 应用很少。

### 2. 各类 xDSL 的特点

#### (1) 速率对称型的 xDSL

比较而言, 对称 xDSL 更适用于企业点对点连接应用, 如文件传输、视频会议等收发数据量大致相同的工作。

HDSL 产生于 20 世纪 80 年代中期, 采用回波抑制、自适应滤波和高速数字处理技术, 线路编码采用 2B1Q 码。用两对或三对模块 (局端机) 和网络终端模块 (用户机), 每对双绞线传输速率 1168Kb/s; 两对线提供 2Mb/s 的 E1 业务。HDSL 无中继传输距离为 3~5km。

美国国家标准协会 (ANSI) 制定的 T1E1.4/94-006 及欧洲电信标准协会 (ETSI) 提出的 DTR/DM-0.3036 定义了 HDSL 的电气及物理特性、帧结构、传输方式及通信规程等标准。由于采用回波抑制自适应均衡技术, 增强了抗干扰能力, 克服了码间干扰, 可实现较

长距离的无中继传输。HDSL 系统分别置于交换局端和用户端,系统由收发器、复用与映射部分以及 E1 接口电路组成。收发器包括发送与接收两部分,是 HDSL 系统的核心。发送部分将输入的 HDSL 单路码流通过线路编码转换,再经过 D/A 转换及波形形成与处理,由发送放大器放大后送到外线。接收部分采用回波抵消器,将泄漏的部分发送信号与阻抗失配的反射信号进行回波抵消,再经均衡处理后恢复原始数据信号,通过线路解码变换为 HDSL 码流,然后送到复用与映射部分处理。其中回波抵消器和均衡器作为系统自适应调整并跟踪外线特性变化,动态调整系统参数,以便优化系统传输性能。

HDSL 系统中复用与映射部分完成数据码流的复用与解复用。发送部分将 2Mb/s 的 PCM 码流分为两部分,分别加入 HDSL 帧结构的开销比特,转换成 HDSL 帧的传输码流,通过发送器发送到线路。与之相对应,接收部分将收到的两路 1168Kb/s 的 HDSL 码流,去掉 HDSL 帧结构中的开销比特,再复用成 2Mb/s 的 PCM 码流,经过接口电路送至 FCM 2 Mb/s 的 G.703 接口。接口部分实现 HDSL 系统与 PCM G.703 接口的转换与适配,完成 HDB3 码流与 NRZ 码流相互转换,并达到电气与阻抗适配。HDSL 的线路编码主要有两种,即 2B1Q 码和 CAP 码。2B1Q 码是 4 电平脉冲幅度调制码,每个符号位表示两比特,4 个电平+3、+1、-1、-3 分别表示 10、11、01、00,从而提高了传输的比特速率。CAP 是无载波幅度相位调制,数据经两路正交信号分别调制后叠加,并且将不携带有用信息的载波抑制掉。两种编码对于提高传输效率是很有用的。目前已有许多实现 HDSL 数字调制及信号处理的芯片和产品问世。

HDSL 除了用于接入网外,还可用于 DDN 节点中继、无线寻呼中继、移动基站中继、局域网间互联、会议电视及高速数据传输等,为企事业用户提供低成本的 E1 通路。利用 HDSL 传输技术可与视频压缩编码 MPEG 技术相结合,传输机频宽带业务,如传输录像机频信号及多媒体会议电视系统信号。按照 H.261 有关建议可作为 2Mb/s 会议电视的传输接入系统。另外,也可用于传输可视电话、家庭购物、远程诊断、远程教育等多媒体业务传输。

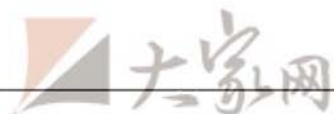
HDSL2 是最新发展起来的利用一对双绞线来提供 T1/E1 速率的 DSL 技术,采用的调制方式和信号处理技术与 HDSL 基本相同。

SDSL 支持对称的 T1/E1 传输,最大有效传输距离为 3km。

## (2) 速率非对称型的 xDSL

ADSL 是近年发展的另一种宽带接入技术,是利用双绞铜线向用户提供两个方向上速率不对称的宽带信息业务。其下行速率高达 6~9Mb/s,传输距离可达 35km,典型的下行速率有 T1、E1、DS2 (6.312Mb/s) 和 E2 (8.448Mb/s)。上行速率低,随各公司产品而不同,通常为 16~640Kb/s。ADSL 在一对电话线上同时传送一路高速下行数据、一路较低速率数据、一路模拟电话。各信号间采用频分复用方式占用不同频带,低频段传送语音;中





间窄频带传送上行信道数据及控制信息；其余高频段传送下行信道数据、图像或高速数据。

ADSL 主要采用 DMT 或 CAP 调制方式。DMT 是离散多频调制，把全部频带划分为 256 个子信道，根据子信道的瞬时衰耗特性、群时延特性和噪声特性，将输入数据动态分配给它们。ADSL 自适应地分配各子信道速率，同时关闭被窄带噪声淹没的子信道，达到传输线的最佳利用，把误码和噪声减至最小，提高了系统传输容量。此外，ADSL 利用自适应滤波的新发展，如栅格编码及 RS 码，提高抗噪声性能，采用非对称回波抵消技术以消除回波干扰。ADSL 系统分别在局端和用户端设置 ADSL Modem，采用 DMT 调制方法，用 IFFT 和 FFT，实现对信号处理及调制与解调。由来自电话网的电话业务，通过普通电话业务 (POTS) 分离器用无源耦合方式接到普通电话线上。交换局 (LEX) 节点经过 ADSL Modem 和 POTS 分离器，分出与插入语音，与高速数据一起通过双绞铜线传到用户端，再经过用户端的分离器与 ADSL Modem，将信息送到用户的 PC 机或电视终端。

按照 ECSA (交换载波标准协会) 的 T1.431 建议，ADSL 可提供高速单工及双工通道，其中高速单工通道利用下行通道传输 6Mb/s 数据流，可同时传输 4 路 MPEG-1 图像，或 1~2 路质量更好的 MPEG-2 图像，或两路 3Mb/s 的数字广播电视。利用 ADSL 的全双工信道可提供 160Kb/s 和 576Kb/s 的数据流，用于接入 ISDN 网或高速链路。ITU-T 提出 3 种 ADSL 标准：ADSL-1、ADSL-2 和 ADSL-3 传输速率分别为 1.544Mb/s、3.152Mb/s 和 6.32Mb/s 传输距离分别为 5.5km、3.6km 和 2.5 km，可用于传输 MPEG-1 和 MPEG-2 图像。

ADSL 技术利用现有双绞线传输宽带业务，可保护投资；降低成本，特别适于大量分散的住宅用户。

它被广泛用于 Internet 接入、远端 LAN 接入、视频点播 (VOD)、远程教学、多媒体检索等宽带业务的接入与传输。目前 ADSL 技术及其应用正不断发展，与之相应的 ADSL 调制与处理芯片也大量推向市场，电信设备厂商也生产出不同类型的 ADSL 调制解调产品并积极开拓市场，使 ADSL 系统在接入网中不断拓宽应用。

RADSL 能提供的速度范围与 ADSL 基本相同，但它可根据铜双绞线质量的优劣和传输距离的远近动态地调整用户的访问速度，目前几乎所有的 ADSL 设备的速率都是自适应的。

ADSL Lite (或 G.Lite) 是 ADSL 的一种简化版本，在 1999 年已被 ITU-T 标准化，它的最高下行速率为 1.536Mb/s，最高上行速率为 512Kb/s。由于最高传输速率有所下降，ADSL lite 设备将有更远的传输距离，而且它取消了用户端设备中的普通电话业务分离器 (splitter)，用户可以自己安装 ADSL 设备，从而降低了成本。

VDSL 与 ADSL 类似，也是非对称型，即利用双绞铜线提供上行与下行两个方向非对称的宽带业务。但 VDSL 的传输速率比 ADSL 高近 10 倍，传输距离比 ADSL 也短得多。VDSL 下行信道的传输速率可达 13~55Mb/s，无中继传输距离为 300m~1.5km；上行信道

传输速率为 1.5~19.2Mb/s, 随不同公司的产品而异。VDSL 也采用频分复用方式, 将电话和 VDSL 的上、下行信号放在不同频带内传输。低频段传输普通电话、ISDN 业务, 中间频段传输上行信道的数字控制信息, 高频段传输下行信道的图像或高速数据信息。发送端将各类业务信号调制到不同频段, 经过双绞线传输到接收端, 接收信号经解调及滤波后分离出各类信号。

VDSL 主要采用 DMT 和离散小波多频调制 (DWMT)。DWMT 采用了小波正交变换, 其性能比 DMT 更好。

经小波变换的 DWMT, 其子信道的功率 99.99% 以上都集中在主瓣, 因而信噪比大为提高。VDSL 系统与 ADSL 类似, 在局端和用户端配置 VDSL Modem, 电话业务通过 POTS 分离器和耦合器加入信道, HDTV 数字图像或多路 MPEG 压缩编码后的图像利用 VDSL 下行信道送至用户端。VDSL 还可与光纤接入网配合, 作为 ONU (光网络单元) 到用户间的配线, 通过 FTTC 为企业用户和家庭提供宽带接入。

各种 DSL 技术的特性比较如表 2-18 所示。

表 2-18 各种 DSL 技术的特性比较

名 称	速率 (Mb/s)	模 式	调 制 方 式	双绞线数目
DSL	0.16	双向		1
HDSL	1.544	双向	2B1Q	1
	2.048	双向		2
SDSL	1.544	双向	2B1Q	1
	2.048	双向		1
ADSL	1~8	下行	CAP/DMT	1
	0.640~1.5	上行		
RADSL	自适应	上行和下行	CAP/DMT	1
VDSL	13~52	下行	CAP/DMT	1
	1.5~2.3	上行		
ADSL Lite (或 UADSL)	1.536	下行	CAP/DMT	1
	0.512	上行		

## 2.5.4 无线以太网基本原理与结构

### 1. WLAN 简介

WLAN (Wireless Local Area Network) 是利用无线通信技术在一定的局部范围内建立的, 是计算机网络与无线通信技术相结合的产物, 它以无线多址信道作为传输媒介, 提供传统有线局域网的功能。WLAN 的覆盖范围一般在 100m 以内, 通过桥接可以达到更大的



覆盖范围。传输介质为红外线 IR 或射频 RF 波段, 以后者使用居多。

## 2. WLAN 的标准

由于 WLAN 是基于计算机网络与无线通信技术的, 在计算机网络结构中, 逻辑链路控制 (Logic Link Contros, LLC) 层及其之上的应用层对不同物理层的要求可以是相同的, 也可以是不同的, 因此, WLAN 标准主要是针对物理层和媒质访问控制层 (Media Access Control, MAC), 涉及到所使用的无线频率范围、空中接口通信协议等技术规范与技术标准。

### (1) IEEE 802.11

1990 年 IEEE 802 标准化委员会成立 IEEE 802.11 WLAN 标准工作组。IEEE 802.11 (又称 Wi-Fi, Wireless Fidelity, 无线保真) 是在 1997 年 6 月由大量的局域网及计算机专家审定通过的标准, 该标准定义了物理层和媒体访问控制 (MAC) 规范。物理层定义了数据传输的信号特征和调制, 定义了两个 RF 传输方法和一个红外线传输方法, RF 传输标准是跳频扩频和直接序列扩频, 工作在 2.4000~2.4835GHz 频段。

IEEE 802.11 是 IEEE 最初制定的一个无线局域网标准, 主要用于解决办公室局域网和校园网中用户与用户终端的无线接入问题, 业务主要限于数据访问, 速率最高只能达到 2Mb/s。由于它在速率和传输距离上都不能满足人们的需要, 所以 IEEE 802.11 标准被 IEEE 802.11b 所取代了。

### (2) IEEE 802.11b

1999 年 9 月 IEEE 802.11b 被正式批准, 该标准规定 WLAN 工作频段在 2.4~2.4835GHz, 数据传输速率达到 11Mb/s, 传输距离控制在 50~150inch。该标准是对 IEEE 802.11 的一个补充, 采用补偿编码键控调制方式, 采用点对点模式和基本模式两种运行模式。在数据传输速率方面可以根据实际情况在 11Mb/s、5.5Mb/s、2Mb/s、1Mb/s 的不同速率间自动切换, 它改变了 WLAN 设计状况, 扩大了 WLAN 的应用领域。

IEEE 802.11b 已成为当前主流的 WLAN 标准, 被多数厂商所采用, 所推出的产品广泛应用于办公室、家庭、宾馆、车站、机场等众多场合, 但是由于许多 WLAN 的新标准的出现, IEEE 802.11a 和 IEEE 802.11g 更是备受业界关注。

### (3) IEEE 802.11a

1999 年, IEEE 802.11a 标准制定完成, 该标准规定 WLAN 工作频段在 5.15~8.825GHz, 数据传输速率达到 54Mb/s/72Mb/s (Turbo), 传输距离控制在 10~100m。该标准也是 IEEE 802.11 的一个补充, 扩充了标准的物理层, 采用正交频分复用 (Orthogonal Frequency Division Modulation, OFDM) 的独特扩频技术, 可提供 25Mb/s 的无线 ATM 接口和 10Mb/s 的以太网无线帧结构接口, 支持多种业务, 如话音、数据和图像等, 一个扇区可以接入多个用户, 每个用户可带多个用户终端。

IEEE 802.11a 标准是 IEEE 802.11b 的后续标准,其设计初衷是取代 IEEE 802.11b 标准,然而,工作于 2.4GHz 频带是不需要执照的,该频段属于工业、教育、医疗等专用频段,是公开的,工作于 5.15~8.825GHz 频带需要执照的。一些公司仍没有表示对 IEEE 802.11a 标准的支持,一些公司更加看好最新混合标准——IEEE 802.11g。

#### (4) IEEE 802.11g

目前,IEEE 推出了最新版本 IEEE802.11g 认证标准,该标准提出拥有 IEEE 802.11a 的传输速率,安全性较 IEEE 802.11b 好,采用两种调制方式,含 IEEE 802.11a 中采用的 OFDM 与 IEEE 802.11b 中采用的 CCK,做到与 IEEE 802.11a 和 IEEE 802.11b 兼容。

### 3. WLAN 网络结构

一般地,WLAN 有两种网络类型:对等网络和基础结构网络。

对等网络由一组有无线接口卡的计算机组成。这些计算机以相同的工作组名、ESSID 和密码等对等的方式相互直接连接,在 WLAN 的覆盖范围的之内,进行点对点与点对多点之间的通信。

在基础结构网络中,具有无线接口卡的无线终端以无线接入点 AP 为中心,通过无线网桥 AB、无线接入网关 AG、无线接入控制器 AC 和无线接入服务器 AS 等将无线局域网与有线网连接起来,可以组建多种复杂的无线局域网接入网络,实现无线移动办公的接入。

### 4. WLAN 接口设计和调试

WLAN 属于无线设备,它的开发和测试都比较复杂。另外,WLAN 芯片生产厂家一般并不详细公开其芯片文档。它们往往需要考核和选择客户,只有客户对它们的芯片用量大到一定规模,它们才会提供给客户详细的资料和参考设计,而且还要收取一笔不菲的技术转让费。

目前,WLAN 中网卡常用的接口有 PCI、MiniPCI、USB、PCMCIA 和 CF 卡接口。嵌入式系统常用的 WLAN 网卡接口一般有 MiniPCI、PCMCIA 和 CF 卡接口。其中 PCMCIA 和 CF 卡接口逻辑类似,而 MiniPCI 接口类似 PCI 接口,只是物理尺寸比较小。

另外需要注意的是,嵌入式系统的 PCMCIA 接口(CF 卡接口也一样)一般都是 16 位模式的,传输速度比较慢,不能支持传输速度较高的 IEEE 802.11g/a 网卡(事实上 PCMCIA 接口的 IEEE 802.11g/a 网卡都是 32 位模式的)。因此如果要在嵌入式系统下支持 IEEE 802.11g/a 网络接口,就必须 MiniPCI 接口。

## 2.5.5 蓝牙接口基本原理与结构

### 1. 概述

蓝牙(Bluetooth)技术是由世界著名的 5 家大公司——Ericsson(爱立信)、Nokia(诺基亚)、Toshiba(东芝)、IBM(国际商用机器公司)和 Intel(英特尔),于 1998 年 5 月联

合宣布的一种无线通信新技术。蓝牙技术的目的是使特定的移动电话、便携式电脑及各种便携式通信设备的主机之间在近距离内实现无缝的资源共享。

蓝牙技术是一种无线数据与语音通信的开放性全球规范,它以低成本的近距离无线连接为基础,为固定与移动设备通信环境建立一个特别连接的短程无线电技术。其实质内容是要建立通用的无线电空中接口(Radio Air Interface)及其控制软件的公开标准,使通信和计算机进一步结合,使不同厂家生产的便携式设备在没有电线或电缆相互连接的情况下,能在近距离范围内具有互用、相互操作的性能。其工作频段为全球通用的 2.4GHz ISM(即工业、科学、医学)频段。蓝牙的数据传输速率为 1Mb/s,采用时分双工方案来实现全双工传输。其理想的连接范围为 10cm~10m,通过增大发送电平可以将距离延长至 100m。

蓝牙基带协议是电路交换与分组交换的结合。在保留的时隙中可以传输同步数据包,每个数据包以不同的频率发送。一个数据包名义上占用一个时隙,但实际上可以被扩展到占用 5 个时隙。蓝牙支持异步数据信道及多达 3 个同时进行的同步话音信道,还可以用一个信道同时传送异步数据和同步话音。每个话音信道支持 64Kb/s,同步话音链路。异步信道可以支持一端最大速率为 721Kb/s 而另一端速率为 57.6Kb/s 的不对称连接,也可以支持 432.6Kb/s 的对称连接。

蓝牙技术的无线电收发器的链接距离可达 30inch,不限制在直线范围内,甚至设备不在同一间房内也能相互链接;并且可以链接多个设备,最多可达 7 个,这就可以把用户身边的设备都链接起来,形成一个个人领域的网络(Personal Area Network, PAN)。

## 2. 蓝牙技术的特点

### (1) 传输距离短

目前蓝牙技术工作距离是 10m 以内,经过增加射频功率后可达到 100m。该工作范围使得蓝牙技术可以保证较高的数据传输速率,同时可降低与其他电子产品和无线电技术的干扰。

### (2) 采用跳频扩频技术

将 2.4GHz~2.4835GHz 之间划分出 79 个频点,采用快速跳频,根据由主机和外设所构成的所谓 Piconet(微网)主单元确定的跳频次数为每秒钟 1600 次。跳频技术的采用使蓝牙的无线链路自身具备了更高的安全性和抗干扰能力。蓝牙空中接口是建立在天线电平为 0dBm 基础上的,空中接口遵循 FCC(美国联邦通信委员会)有关电平为 0dBm 的 ISM 频段的标准。如果全球电平达到 100mW 以上,还可以使用扩展频谱功能来增加一些补充业务。频谱扩展功能是通过起始频率为 2.402GHz、终止频率为 2.480GHz、间隔 1MHz 的 79 个跳频频点来实现的。出于对某些本地规定的考虑,日本、法国和西班牙都缩减了带宽。最大的跳频速率为 1600 跳/s。

### (3) 采用时分复用多路访问技术

蓝牙的基带符号速率为 1Mb/s, 它采用数据包的形式按时隙传送, 每时隙 0.625ms, 不排除将来采用更高的符号速率。每个蓝牙设备均在自己的时隙中发送数据, 这在一定程度上有效地避免了无线通信中的“碰撞”和“隐藏终端”等问题。

#### (4) 网络技术

几个 Piconet 可以被连接在一起, 并依靠跳频顺序识别每个 Piconet。同一 Piconet 的所有用户都与这个跳频顺序同步, 其拓扑结构可以被描述为多 Piconet 结构。在一个由 10 个独立的全负载 Piconet 组成的多 Piconet 结构中, 全双工数据速率可超过 6Mb/s。

#### (5) 语音支持

语音信道采用 CVSD (连续可变斜率增量调制) 语音编码方案, 且从不重发语音数据包。CVSD 编码擅长处理丢失和被损坏的语音采样, 即使错误率达到 4%, 经过 CVSD 编码处理的语音同样可以被识别。

#### (6) 纠错技术

蓝牙技术采用的是 FEC (前向纠错) 方案, 其目的是为了减少数据重发的次数, 降低数据传输负载。但是, 要实现数据的无差错传输, FEC 就必然要生成一些不必要的开销比特而降低数据的传送效率。这是因为, 数据包对于是否使用 FEC 是弹性定义的。报头总有占 1/3 比例的 FEC 码起保护作用, 其中包含了有用的链路信息。在无编号的 ARQ 方案中, 一个时隙中传送的数据必须在下一个时隙得到确认。只有数据在接收端通过了报头错误检测和循环冗余检测后认为无错, 才向发送端返回确认消息, 否则将返回一个错误消息。

### 3. 蓝牙接口的组成

蓝牙接口主要由 3 大单元组成。

- 无线单元: 主要完成基带信号和射频信号之间的上下转换, 实现数据流的过滤和传输。
- 基带单元: 主要完成跳频控制, 数据和信息的打包与传输。
- 链路管理与控制单元: 是系统的核心部分, 它负责连接的建立和拆除以及链路的安全和控制, 还要为上层软件提供访问入口。

### 4. 链路管理与控制

在 Piconet 内的连接被建立之前, 所有的设备都处于旁观 (standby) 状态。此时, 这些设备周期性地“监听”其他设备发出的查询 (inquire) 消息或寻呼 (page) 信息。首先请求连接的单元是 master 单元, 如果对方地址已经存在于 master 单元的地址簿中, master 单元则通过发出寻呼 (page) 消息包请求建立连接; 如果地址未知, 则首先通过查询 (inquire) 消息包查询覆盖范围内其他单元的地址, 然后再用寻呼 (page) 消息包建立连接。

在查询过程中, master 单元使用特别预留的全球统一的 Inquire 事件 ID 号, 并采用全球唯一的包含 32 个信道的信道序列发送此指令。Inquire scan 的设备周期性地在这 32 个信



道上进行监听,直到该设备的 Inquire scan 功能被禁止。在主单元端将这 32 个信道分为两组,每组 16 个,主单元先在第一组的 16 个信道上发布 Inquire 指令,随即在回复信道上进行监听,如果被查询单元接收到 Inquire 指令,则用 FHS 包发送自己的 ID 号和时钟偏移;然后主单元在第二组的 16 个信道上发布 Inquire 指令,如此反复,直到覆盖范围内所有单元都发回 FHS 包,主单元就建立了一个完整的覆盖范围内的设备情况表。

在寻呼过程中,32 个寻呼信道也分为两组,主单元根据它最近知道的被呼单元的 ID 号和时钟偏移来调整两个信道组的频率,然后先用第一组频率持续呼叫 1.25 s。如果没有收到回音,则再用第二组频率持续呼叫 1.25s。被呼单元则每隔 1.25s 在寻呼信道中监听寻呼消息中的 ID 信息,一旦发现接收到的 ID 号与本身的 ID 号一致,则发送回复消息包。这样,微网就形成了。

如果 Piconet 中已经处于连接的设备在较长一段时间内没有数据传输, master 可以把 slave 置为 hold 模式,这时,只有一个内部计数器工作。slave 也可以主动要求被置为 hold 模式。一旦处于 hold 模式的单元被激活,则数据传递也立即重新开始。hold 模式一般被用于连接好几个微网的情况或者耗能低的设备,如温度传感器。除了 hold 模式外,蓝牙还支持 sniff 模式和 park 模式两种节能工作模式:在 sniff 模式下,slave 降低了从 Piconet“收听”消息的速率,“呼吸”间隔可以依应用要求做适当调整;在 park 模式下,设备依然与 Piconet 同步但没有数据传送。工作在 prk 模式下的设备放弃了 Ama 地址,偶尔“收听”master 的消息并恢复同步、检查广播消息。采用 park 模式可以使一个微网中的 master 单元管理的 slave 单元数远远大于 7 个。如果把这几种工作模式按照节能效率以升序排列,则依次是:呼吸模式、保持模式和暂停模式。

在活动状态下,蓝牙技术支持两种链路类型:SCO(面向连接的同步链路)和 ACL(面向无连接的异步链路)。

SCO 链路为对称连接,利用保留时隙传送数据包。它可以被认为是一种主单元和从单元之间的电路交换的点对点连接,主要用于传送实时语音,也可以传送数据,但在传送数据时,只用于重发被损坏的那部分数据。每个蓝牙单元最多可同时支持 3 个 SCO 链路。

ACL 链路既支持对称连接,也支持不对称连接。master 负责控制链路带宽,并决定微网中的每个 slave 可以占用多少带宽和连接的对称性。slave 只有被选中时才能传送数据。ACL 链路也支持 master 发给微微网中所有 slave 的广播消息。ACL 链路一般不能传送语音,但可以传送 IP 语音。

## 5. 蓝牙接口的主要应用

从理论上讲,蓝牙技术可以被植入到所有的数字设备中,用于短距离无线数据传输。目前可以预计的应用场所主要是计算机、移动电话、工业控制及 PAN 的连接。

### (1) 蓝牙在计算机中的应用

蓝牙接口可以直接集成到计算机主板,也可通过 PC 卡或 USB 接口连接,实现计算机之间及计算机与外设之间的无线连接。这种无线连接对于便携式计算机可能更有意义。通过在便携式计算机中植入蓝牙技术,便携式计算机就可以通过蓝牙移动电话或蓝牙接入点连接远端网络,可以方便地进行数据交换。当便携式计算机中的某些资料更新后,可以在不需人工干预的情况下,对家用台式计算机进行同步更新。

### (2) 蓝牙在移动电话中的应用

从目前来看,移动电话是蓝牙技术的最大应用领域,也是已经有实际应用的领域。通过在移动电话中植入蓝牙技术,可以实现无线耳机、车载电话等功能,还能实现与便携式计算机和其他手持设备的无电缆连接,组成一个方便灵活的 PAN。当蓝牙技术普及后,蓝牙移动电话还能作为一个工具,实现所有的商用卡交易。

### (3) 蓝牙在其他方面的应用

目前,蓝牙技术的应用领域已越来越广阔,例如,将蓝牙技术应用于汽车行业中,实现对汽车各部件的实时监控;将蓝牙技术应用于建筑行业中,实现智能化住宅;在人口密度大的地区(如车站、机场、商场等)为用户提供接入服务等。

## 2.5.6 1394 接口基本原理与结构

### 1. IEEE 1394 简介

IEEE 1394 是美国 Apple 公司率先提出的一种高品质、高传输速率的串行总线技术。1995 年被 IEEE 认定为串行工业总线标准,命名为 1394-1995,后来又在其基础上增加了被称为 1394a 的附加规范。近年又计划提出新的 1394b 规范。世界几大计算机公司包括 IBM、Apple、Microsoft 等都支持这种总线。虽然目前多数计算机不含 1394 的接口,但越来越多的迹象表明,1394 将成为一种新的串行总线标准,得到广泛使用。

IEEE 1394 本来主要应用于实时多媒体领域,例如消费电子应用,如数码摄像机、DVD、数字 VCRs 以及音乐系统。在 PC 机上,它主要用于大容量存储器以及打印机、扫描仪之上,作为这些设备的数字化高速接口。由此看出,IEEE 1394 作为一种标准总线,可以在不同的工业设备间架起一座沟通的桥梁。采用 IEEE 1394 的典型意义在于,在一条 IEEE 1394 总线上可以接入 63 个设备,大大减少了外设接口的数量。

### 2. IEEE 1394 的特点

IEEE 1394 的主要特点包括:

- 支持多种总线速度,适应不同应用要求。IEEE 1394a 支持的速度范围为 100Mb/s, 200Mb/s 和 400Mb/s,其中支持 100Mb/s 和 200Mb/s 的总线设备已经推出。IEEE1394b 支持的速度更高,为 800Mb/s, 1600Mb/s 和 3200Mb/s。不像 USB,

在一个 IEEE 1394 系统中, 各种速度的设备可以共存, 但不互相影响通讯速度。

- 即插即用, 支持热插拔。在任何时刻, 用户均可以将设备加入到总线中或从总线中移去而不必关掉电源或重新启动计算机。总线控制器会自动重新配置好设备。每个设备的资源均由总线控制自动分配, 用户不作任何繁琐的配置工作。
- 支持两种传输方式, 即同步和异步的传输方式。设备可以根据需要动态地选择传输方式, 总线自动完成带宽分配。异步传输方式类似于内存映射 I/O 总线方式, 此时, 它类似于 PCI 总线, 任何设备可以在 64 位的地址空间内进行读写操作。异步传输实际上是由一个发送应答行为组合而成的, 对于不同的传输速度, 异步包的大小是不一样的。同步传输方式下, 一个重要的特点是总线控制器为每次传输保证足够的带宽。它不是用地址空间而使用频道进行数据的传输。每个频道有一个频道号, 发送方和接收方在指定号的频道上进行数据的读写操作。同步周期为  $125\mu\text{s}$ 。两种传输方式可以适用不同的传输要求: 在要求实时传输并对数据的完整性要求不严格的场合, 可采用同步传输方式。如果对数据的完整性要求较高的话, 采用异步传输方式更好。
- 支持点到点通信模式。和 USB 的主从式结构不同, IEEE 1394 是多主总线 (类似于 PCI), 每个设备都可以获取总线的控制权, 与其他设备进行通信。这使设备的直接互连成为可能。
- 遵循 ANSI IEEE 1212 控制及状态寄存器 (CSR) 标准, 该标准定义了 64 位的地址空间, 可寻址 1024 条总线的 63 个节点, 每个节点可包含 256TB 的内存空间。
- 支持较远距离的传输。普通线缆环境下, 两个设备之间的最大距离可达 4.5 m (高级线缆可达 15 m), 使用中继器可以延长两个设备间的距离至 72 m, 跨越最多 16 个中继器。IEEE 1394b 规范支持多介质传输, 用玻璃光缆或 5 类双绞线传输, 设备间距离可达 100 m 以上。
- IEEE 1394 支持公平仲裁原则, 为每一种传输方式保证足够的传输带宽。同时, 支持错误检测和处理。
- IEEE 1394 六线电缆具有电源线, 可传输 8~40V 的直流电压, 某些特定的节点可通过电源线向总线供电, 其他节点可以从总线获取能量。

### 3. IEEE 1394 的协议结构

IEEE 1394 总线的物理拓扑结构包括两种类型, 一种是总线底板结构, 另一种是电缆结构, 这两种结构之间要通过总线桥连接。这里将主要介绍采用电缆环境的 IEEE 1394。从物理结构来讲, IEEE 1394 的电缆是由 6 根屏蔽双绞线组成, 其中两对双绞线用于信号的传递, 另外一对用于供应电源。IEEE 1394a 标准的附录中还规定了一种 4 根导线的可选



无源连接电缆。

IEEE 1394 的协议栈由 3 层组成：物理层、链路层及事务层。另外还有一个管理层。物理层和链路层由硬件构成，而事务层主要由软件实现，如图 2-60 所示。

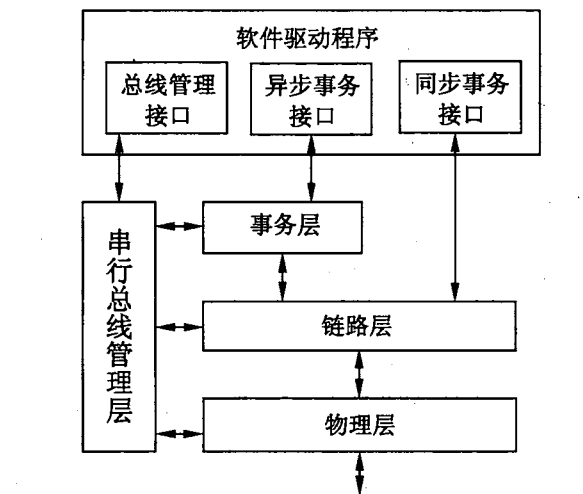


图 2-60 IEEE 1394 的分层结构模型

物理层提供了 IEEE 1394 的电气和机械接口，它的功能是重组字节流并将它们发送到目的节点上去。同时，物理层为链路层提供服务，解析字节流并发送数据包给链路层。

链路层提供了给事务层确认的数据包服务，包括：寻址、数据组帧及数据校验。链路层还提供直接面向应用的服务，包括产生  $125\mu\text{s}$  的同步周期。链路层支持两种传输模式。链路层的底层（对应于 OSI 的介质访问层，也有的书上将它归为物理层）提供了仲裁机制，以确保同一时间上只有一个节点在总线上传输数据。

事务层为应用提供服务。它定义了 3 种基于请求响应的服务，分别为 read、write 和 lock。事务层只支持异步传输。同步传输服务由链路层提供。

管理层定义了一个管理节点所使用的所有协议、服务以及进程。电缆环境下，IEEE 1394 定义了两大类管理：总线管理（BM）和同步资源管理（IRM）。BM 包含总线的电源管理信息、拓扑结构信息及不同节点的速度极限信息，以便协调不同速度设备之间的通信。IRM 管理同步资源，如可用频道信息以及带宽的分配。可充当管理层角色的节点是可选的。

一般来讲，物理层及链路层由电路来实现，通常集成在同一芯片上。而事务层是由软件来实现的。由此可以看出，IEEE 1394 定义了分层的协议结构，这使它更适合于网络应用，通过各协议层的配合工作，可以为终端用户提供可靠、快速的通信服务。



## 2.6 嵌入式系统电源

### 2.6.1 电源接口技术

所有嵌入式系统设计都必须包含电源，可以选择 AC 电源插座或电池供电。下面对这两种方式及电源的稳压进行介绍。

#### 1. AC 电源

如果嵌入式系统对便携性没有太高的要求，那么使用来自插座的电源是最佳的供电方式。但因为交流电电压很高，不能直接用于嵌入式系统，还需要转化为电压低得多的直流电。可以使用实验室直流电、标准 PC 电源或交流电适配器。其中，交流电适配器对于大多数应用来说可能是最好的选择。

交流电适配器的外形就是一个小的黑盒子，它可以对嵌入式系统进行供电。这种解决办法价格便宜、使用方便，且可靠性较高，从电子设备商那里可以购买到交流电适配器，它通常提供+5V DC~+12V DC 不等的输出电压，提供的电流可以高达 500mA，具体的电压和电流取决于系统的需求，从中挑选满足电压和电流要求的交流电适配器即可。另外，注意连接器的极性，有些交流适配器的正极电压位于连接器插座的中央，而接地在外面；而有些正好相反。因此，在对嵌入式系统供电前，需要弄清楚适配器的极性，否则可能造成灾难性的后果。

#### 2. 电池

电池使用方便，容易携带，但需要选择合适的电压和足够的电流。只有电池选择恰当、系统设计合理，才能保证嵌入式系统的正常工作。例如，一个很小的基于 PIC 或 AVR 的微处理器系统能够运转两年之久而仅仅耗费一节 AA 电池，而当设计不合理时，系统则会在几分钟内用完一节 AA 电池。电池选择不当不仅不能为系统提供足够的电流，而且会导致系统操作不稳定，甚至使系统根本就无法启动。选择电池时，不仅要考虑其平均电流量，还要考虑到其峰值电流。因为一个嵌入式系统平时可能只需要 20mA 的恒定电流，但它在峰值负载时需要 100mA 的电流。对于使用 Flash Memory 的嵌入式系统更是如此，因为 Flash Memory 在写操作过程中需要较高的电流。这种系统的电池不仅要能够在负载恒定时给系统供电，也必须能够在峰值负载时给系统供电。

#### 3. 稳压器

稳压器是一个把输入的 DC 电压（通常为一个输入电压的范围）转换为固定输出 DC 电压的半导体设备，它主要用来为系统提供恒定的电压。

虽然嵌入式系统里的很多元件在一个很宽的电源范围内都可以运转，但一个固定的工

作电压对系统的正常工作还是很重要的，如 A/D 转换器。因为很多设备使用这一内部电压作为参考电压。

另外，稳压器有助于去除电源的噪声，给从外部电源供电的嵌入式系统提供了一定程度的保护和隔离。如果系统靠电池供电运行，那么系统变化着的电流，结合电池内部阻抗的作用，将产生一个变化的电压，使嵌入式系统不能正常工作；稳压器就可以防止这种问题的发生。下面介绍 DC-DC 转换器的稳压器类型，它可以接收不稳定的 DC 电压，而输出一个恒定电压值的稳定 DC 电压。

DC-DC 转换器有 3 种类型：

- 线性稳压器，产生较输入电压低的电压。
- 开关稳压器，能升高电压、降低电压或翻转输入电压。
- 充电泵，可以升压、降压或翻转输入电压，但电流驱动能力有限。

任何变压器的转换过程都不具有 100% 的效率，稳压器本身也使用电流（称为静态电流），这个电流来自输入电流。静态电流越大，稳压器的功耗越大。在选择稳压器时，应尽量选择既能满足嵌入式系统电压和电流的要求，又保证静态电流低的变压器。

线性稳压器体积小，价格便宜、噪声小且使用方便，其输入输出使用退耦电容来过滤，电容除了有助于平稳电压以外，还有利于去除电源中的瞬间短时脉冲波形干扰。电源的这种瞬间变弱很少发生，但一旦发生就会严重影响到系统的运行。许多嵌入式微处理器包含电压不足检电器，一旦电源输入给处理器的电压过低，检电器就会重启处理器。

开关稳压器由于其输出端开关功率管（Metal-Oxide Semiconductor Field Effect Transistor, MOSFET）是一种高输入阻抗、低开关速度及低功耗得半导体器件而得名。在变换输入电压为输出电压时，开关稳压器得功耗更低、效率更高。其缺点为需要较多的外部器件，如电感和二极管等，因而占用的空间较大。开关稳压器比线性稳压器要贵，而且产生的噪声也大，但它可以升压、降压和翻转电压，它的功能比线性稳压器强大。

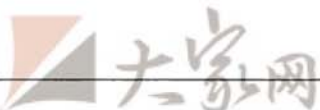
与开关稳压器相似，充电泵能够升压、降压和翻转输入电压；不同之处在于它不需要外部电感。但由于其电流供应能力有限，因此很少使用。

## 2.6.2 电源管理技术

对于绝大多数的便携式设备而言，低功耗和节能设计都是最重要的设计指标。例如，笔记本电脑依靠电池供电时，就会进入低功耗工作模式；PDA 停止使用一段时间后显示屏将变暗，设备甚至进入睡眠状态。便携式设备的低功耗设计是通过电源管理技术实现的。

### 1. 电源管理技术

首先，操作系统除了能简单地给嵌入式处理器内核启用空闲模式之外，还需要担任更多的电源管理支持。在实践中，大量功率被周边设备所消耗，可能是片上器件，也可能是



外部设备, 此外存储器也会消耗大量功率。任何电源管理方法都应当具备管理外设功耗的支持, 这是至关重要的。此外, 电压与功耗之间的平方关系意味着理想高效的方法是在要求较低电压的较低时钟速率上执行代码, 而不是先以最高的时钟速率执行然后再转为空闲。这里介绍一下在嵌入式系统中常用的电源管理技术。

### (1) 系统上电行为

微处理器及其片上外设一般均以最高时钟速率上电启动。但是, 有些资源的供电启动还尚不需要, 或者根本就不会在应用过程中用到。例如, MP3 播放器就很少使用其 USB 端口与 PC 进行通信。在启动时, 系统必须为应用提供一种调节系统的机制, 从而关闭不必要的电源消耗器件或使之处于空闲状态。

### (2) 空闲模式

CMOS 电路中的有效功耗只有在电路进行时钟计时的情况下才发生。通过关闭不需要的时钟, 可以消除不必要的有效功耗。在等待外部事件时, 大多数微处理器都融入了暂时终止 CPU 有效功耗的机制。CPU 时钟的“闲置”通常由“停止”或“闲置”指令触发, 其在应用或操作系统闲置时进行调用。一些 DSP 进行多个时钟域分区, 可以使这些域分别处于空闲状态, 以中止未使用模块中的有效功耗。例如, TI 的 TMS320C5510 DSP 中, 可以有选择性地使 6 个时钟域闲置, 其中包括 CPU、cache、DMA、外设时钟、时钟生成器及外部存储器接口。

除了支持闲置 DSP 及其片上外设之外, 嵌入式系统还必须提供用于闲置外部周边设备的机制。例如, 一些编码译码器具备可以被激活的内置低功率模式。我们面临的一个挑战是类似看门狗定时器这样的外设。通常情况下, 看门狗定时器应根据预定义的时间间隔提供服务, 以避免其激活。这样, 减缓或中止处理的电源管理技术就可能无意中导致应用故障。因此, 该嵌入式系统应当使应用在睡眠模式期间禁用此类外设。

### (3) 断电

尽管空闲模式消除了有效功耗, 但静态功耗即便在电路不进行切换的情况下也会出现, 这主要是由于逆向偏压泄漏 (reverse-bias leakage) 造成的。如果系统包括的某个模块不必随时供电, 那么就可以让系统仅在需要时才为子系统上电, 从而减少功耗。到目前为止, 嵌入式系统开发商对最小化静态功耗投入的工作极少, 因为 CMOS 电路的静态功耗非常低。但是, 新型、具有更高性能的晶体管使电流泄漏显著增加, 这就要求用户对可降低静态功耗及更复杂的睡眠模式给予新的关注。

### (4) 电压与频率缩放 (frequency scaling)

有效功耗与切换频率成线性比例, 但与电源电压平方成正比。经较低频率运行应用与在全时钟频率上运行该应用并转入闲置相比, 并不能节约多少功率。但是, 如果频率与平台上可用的更低操作电压兼容, 那么就可以通过降低电压来大大节约功耗, 这正是因为

存在上述平方关系的缘故。

在嵌入式系统中的低功耗设计主要注意如下问题：

- 系统中 CPU 以外的其他电路器件尽可能选用静态功耗低的器件，如选用 CMOS 电路芯片。
- 外部设备的选择也要尽可能支持低功耗设计。
- 设计外部中断唤醒电路，使 CPU 在等待时可进入休眠模式或待机模式，需要时由外部中断信号唤醒。
- 设计外部器件的电源控制电路，当外部器件或设备在不工作时关断供电，减少无效功耗。
- 使用充分降低系统功耗的软件。

因此，电源通常被认为是整个系统的“心脏”，绝大多数电子设备 50%~80% 的节能潜力在于电源系统。研制开发新型开关电源是节能的主要举措之一。近年来许多公司相继推出一系列功能齐全、种类繁多的低功耗器件，具有种类更多、功耗更低、体积更小、使用方便等特点。

## 2. 降低功耗的设计技术

电路的设计与元件的选取是同时或交叉进行的。在功能要求相同的情况下，不同的人可以设计出不同的电路，虽然功能可以相同，但电路功耗却往往相距甚大。电路设计和元件选取考虑的因素很多，对其中需要注意的地方进行介绍。

### (1) 采用低功耗器件

几乎所有的 TTL 工艺的逻辑电路、单片机、存储器以及外围电路都有相应 CMOS 工艺的低功耗器件，采用这些器件是降低系统功耗最直接的方法。

### (2) 采用高集成度专用器件

例如用单片机设计一个电子体温计，就没有必要采用 80C51 单片机，而应该采用 Epson、Holtek 等生产的专用于测量体温的单片机，其内部集成了测量体温所需要的 ADC、振荡器、电压基准、LED 显示驱动等部件，电路只需几个电阻电容元件整个电路可在 1.2~1.5V 电压下工作，功耗极低，而且可靠性、体积等都比用分离器件设计更好。DC/DC 变换器在市场上有各种各样的模块供选择，而且效率高、功耗低、体积小、可靠性高，完全没必要采用分离电路搭接。

### (3) 动态调整处理器的时钟频率和电压

在系统指标允许的情况下，尽量使用低频率器件有助于降低系统功耗。处理器根据当前的工作负载，运行在不同的性能等级上。例如，一个 MPEG 视频播放器需要的处理器性能比 MP3 音频播放器高一个数量级。因此，当播放 MP3 时，处理器可以运行在较低频率上，而仍然能保证播放的高质量。当时钟频率降低时，可以同时降低处理器的供电电压，

以达到节能的目的。

动态电压调整技术(DVS)就利用了CMOS工艺处理器的峰值频率与供电电压成正比这一特点。减少供电电压并同时降低处理器的时钟速度,功耗将会呈二次方的速度下降,代价是增加了运行时间。

#### (4) 利用“节电”工作方式

许多器件都有低功耗的“节电”方式,如微处理器的闲置、掉电工作方式,存储器的维持工作、ADC和DAC的节能工作方式等,因此设计时充分利用其“节电”方式为达到节电的效果。

另外,合理处理器件的空余引脚也是非常重要的。大多数数字电路的输出端在输出低电平时,其功耗远大于输出高电平时的功耗,设计时应注意控制低电平的输出时间,闲置时使其处于高电平输出状态。因此,多余的非门、与非门的输入端应接低电平,多余的与门、或门的输入端应接高电平。对ROM或RAM及其他有片选信号的器件,不要将“片选”引脚直接接地,避免器件长期被接通,而应与“读/写”信号结合,只对其进行读或写操作时才选通器件。

#### (5) 实行电源管理

目前大部分的传感器、接口器件、显示器件等本身还没有低功耗工作模式,而有些便携式仪器又不可避免地要使用它们,这些器件往往成了电路中的“耗电大户”。这种情况下,可对电路进行模块设计,工作时对大功耗模块实施间断供电,即设置电源形状电路,并通过软件或定时电路控制开关,使大功耗模块电路仅在需要工作的短时间内加电,其余时间则处于断电状态。

现在便携式电子产品对供电电路的要求越来越复杂,不仅要求电源本身稳定,而且还要求有电压监测、电源管理功能,还要满足小型化、延长电池寿命等要求。便携式产品设备由于受尺寸、成本的限制,往往在着手设计供电电路之前就已经确定了电池的数量和种类。电池数量限制了电源的电压范围,直接影响电源管理电路的成本和复杂程度。对于电池节数多的系统可选用线性稳压器,电路设计简单、成本低,但转换效率相对较低;对于电池节数少的系统则须选用成本较高的开关电源,电路设计复杂,但由于减少了电池数量,电源成本可降低。

由于便携式嵌入式系统的设计需要考虑尺寸、重量、成本、电池种类、转换效率(电池工作时)等诸多因素,不同产品对以上指标的要求会有不同的侧重。例如,只是偶尔处于工作状态的产品较注重电源在空载时的静态电流,并不十分注重满荷下电源的工作效率;蜂窝电话则注重电源所能提供的峰值电流和转换效率。因此,很难研制出一种电源芯片适应所有产品的需求,嵌入式系统的多样化导致了电源芯片的多样化。

一般系统中常含有数字处理系统、模拟处理系统及其他系统通信的各种数字通信接

口、模拟通信接口、人机交换接口、与传感器及执行机构连接的模拟接口等。这些单元常用的电源种类归纳如表 2-19 所示。

表 2-19 常用电源及用途

电 源 种 类	主 要 用 途
+5V	数字处理系统主电源
-5V	运放、ADC、DAC、LCD 偏压电路
+3.3V、+2.0V、+1.8V	低电压逻辑、CPU、DSP、FPGA 等内核电路
+12V、+15V、-12V、-15V	传感器、模拟处理系统、运放、ADC、DAC、通信接口
+5V	传感器、通信接口、光盘耦合器
+24V	4~20mA 模拟接口、温湿度控制器、一些执行机构、蓄电池

小功率电源管理芯片有 AD/DC 和 DC/DC 两大类。单片 AC/DC 电源变换器属于无源变压器的小功率一体化线性稳压电源，使用极其方便。DC/DC 电源变换器可分为压式变换器、降压式变换器、极性反转/倍压式变换器、低压差集成线性稳压器等。

## 2.7 电子电路设计基础

### 2.7.1 电路设计

#### 1. 电路设计原理

电路板的设计主要分 3 个步骤：设计电路原理图、生成网络表、设计印制电路板，如图 2-61 所示。

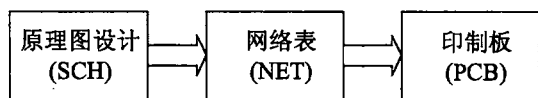


图 2-61 电路板设计的 3 个步骤

进行硬件设计开发，首先要进行原理图设计，需要将一个个元器件按一定的逻辑关系连接起来。设计一个原理图的元件来源是“原理图库”，除了元件库外还可以由用户自己增加建立新的元件，用户可以用这些元件来实现所要设计产品的逻辑功能。例如利用 Protel 中的画线、总线等工具，将电路中具有电气意义的导线、符号和标识根据设计要求连接起来，构成一个完整的原理图。

原理图设计完成后要进行网络表输出。网络表是电路原理设计和印制电路板设计中的一个桥梁，它是设计工具软件自动布线的灵魂，可以从原理图中生成，也可以从印制电路

板图中提取。常见的原理图输入工具都具有 Verilog/VHDL 网络表生成功能, 这些网络表包含所有的元件及元件之间的网络连接关系。

原理图设计完成后就可进行印制电路板设计。进行印制电路板设计时, 可以利用 Protel 提供的包括自动布线、各种设计规则的确定、叠层的设计、布线方式的设计、信号完整性设计等强大的布线功能, 完成复杂的印制电路板设计, 达到系统的准确性、功能性、可靠性设计。

## 2. 电路设计方法 (有效步骤)

电路原理图设计不仅是整个电路设计的第一步, 也是电路设计的基础。由于以后的设计工作都是以此为基础, 因此电路原理图的好坏直接影响到以后的设计工作。电路原理图的具体设计步骤, 如图 2-62 所示。

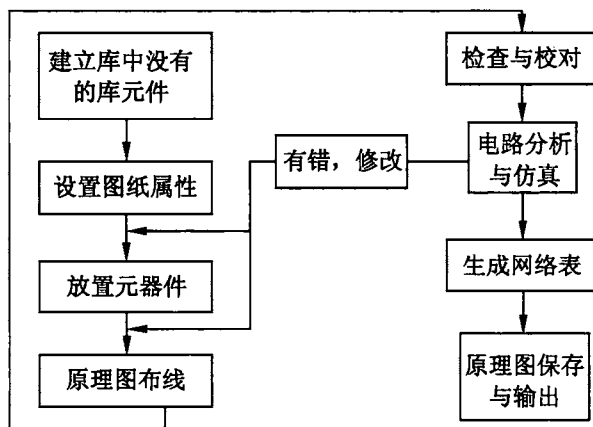


图 2-62 原理图设计流程图

### (1) 建立元件库中没有的库元件

元件库中保存的元件只有常用元件。设计者在设计时首先碰到的问题往往就是库中没有原理图中的部分元件。这时设计者只有利用设计软件提供的元件编辑功能建立新的库元件, 然后才能进行原理图设计。

当采用片上系统的设计方法时, 系统电路是针对封装的引脚关系图, 与传统的设计方法中采用逻辑关系的库元件不同。

### (2) 设置图纸属性

设计者根据实际电路的复杂程度设置图纸大小和类型。图纸属性的设置过程实际上是建立设计平台的过程。设计者只有设置好这个工作平台, 才能够上面设计符合要求的电路图。

### (3) 放置元件

在这个阶段,设计者根据原理图的需要,将元件从元件库中取出放置到图纸上,并根据原理图的需要进行调整,修改位置,对元件的编号、封装进行设置等,为下一步的工作打下基础。

### (4) 原理图布线

在这个阶段,设计者根据原理图的需要,利用设计软件提供的各种工具和指令进行布线,将工作平面上的元件用具有电气意义的导线、符号连接起来,构成一个完整的原理图。

### (5) 检查与校对

在该阶段,设计者利用设计软件提供的各种检测功能对所绘制的原理图进行检查与校对,以保证原理图符合电气规则,同时还应力求做到布局美观。这个过程包括校对元件、导线位置调整以及更改元件的属性等。

### (6) 电路分析与仿真

这一步,设计者利用原理图仿真软件或设计软件提供的强大的电路仿真功能,对原理图的性能指标进行仿真,使设计者在原理图中就能对自己设计的电路性能指标进行观察、测试,从而避免前期问题后移,造成不必要的返工。

### (7) 生成网络表

这一步,设计者利用设计软件提供的网络表生成工具,建立起该原理图的网络表。其实每个电路就是一个网络表,它是由节点、元件和连线组成的。电路原理图的网络表是电路板自动布线的灵魂,也是原理图设计软件与印刷电路设计软件之间的接口。

### (8) 保存与输出

这一步是设计者对设计好的原理图进行存盘,输出打印,以供存档。这个过程实际是一个对设计的图形文件输出的管理过程,是一个设置打印参数的过程。

## 2.7.2 PCB 电路设计

### 1. PCB 设计原理

原理图设计完成后就可以进行印制电路板(Printed Circuit Board, PCB)设计了。PCB是电子产品的基石。任何电子产品都是由形形色色的电子元件组成,而这些电子元件的载体和相互连接所依靠的正是印制电路板。不断发展的PCB技术使电子产品设计和装配走向标准化、规模化、自动化,并使得电子产品体积减小,成本降低,可靠性和稳定性提高,装配、维修简单。可以这样说,没有PCB,就没有现代电子信息产业的高速发展,就没有今天的电子信息技术。

PCB是由印制电路加上基板构成的。对于PCB的材料,以及PCB的制作工艺,不是本书关心的部分,读者可以通过工艺方面的参考书获得。下面了解一下PCB相关的概念。



- 印制：采用某种方法，在一个表面上再现图形和符号的工艺，它包括通常意义的印刷。
- 印制线路：采用印制法在基板上制成的导电图形，包括印制导线、焊盘等。
- 印制元件：采用印制法在基板上制成的电路元件符号。
- 印制电路：采用印制法得到的电路。
- 印制电路板：完成了印制电路和印制线路加工的板子。
- 印制电路板组件：安装了元器件或其他部件的印制电路板。

对于 PCB 的分类，有很多种方法。按照 PCB 的层数来分，一般分为单面板、双面板和多层板；按照机械性能区分，一般分为刚性板、柔性板；按基材材料区分，可分为纸基板、玻璃布基板、复合材料基板和特种材料基板等。一般电子电器、通信雷达和大型通信产品的 PCB 多是刚性、多层玻璃基材料板。一些手机终端或小型电子设备采用柔性板。

## 2. PCB 设计方法（有效步骤）

PCB 的设计是电子产品物理结构设计的一部分，它的主要任务是根据电路的原理和所需元件的封装形式进行物理结构的布局和布线。具体步骤如图 2-63 所示。

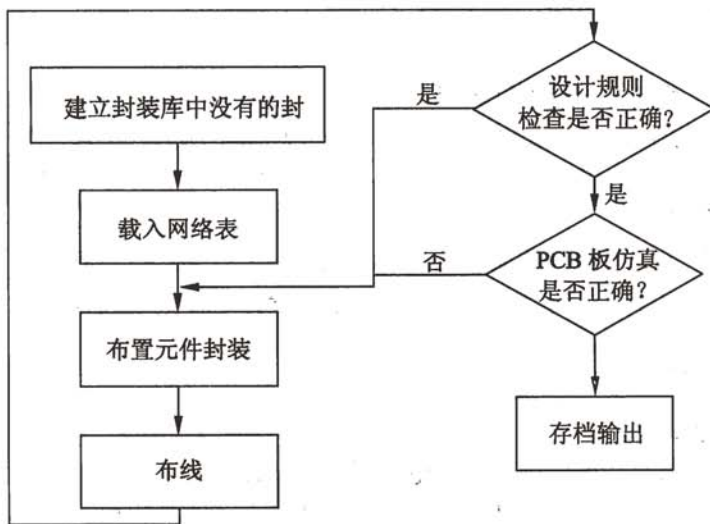


图 2-63 PCB 设计流程图

### （1）建立封装库中没有的封装

封装库里保存的只有一些常用元件的封装，设计者在设计 PCB 时，通常首先遇到的问题就是在封装库中找不到合适的封装，这时只能先利用设计工具（如 Protel 99 SE）提供的元件封装编辑器新建该元器件的封装。总之，在设计相应的 PCB 图之前，先要保证所用的

元件的封装在封装库中是齐全的。

### (2) 规划电路板

在封装库准备好以后,设计 PCB 的第一步是规划电路板。规划包括以下内容:设置习惯性的环境参数和文档参数,如选择层面、外形标尺大小等。

### (3) 载入网络表和元件封装

在规划好电路板以后,就要以载入前面所准备的网络表,将元件封装自动放入电路规划的外形范围内。但这些元件封装是叠放在一起的,设计者必须将它们分开,并放置在适当的位置。

### (4) 布置元件封装

元件封装的布置可采用自动布置和手工布置结合的方法,将元件封装放置在适当的位置。这里的“适当”包含两个意思:一是使元件放置在让人满意的位置,将元件布置得整齐美观;二是使元件放置在有利于布线的位置。

### (5) 布线

在元件布置完成后,可设置设计规划,开始自动或手工布线了。在采用自动布线时,如果布线没有完全成功,或者有不满意和出现违规错误的地方,就要进行手工调整。

### (6) 设计规则检查

设计的 PCB 板图是由许多图件构成的,如元件、铜箔线、过孔等,在旋转多个图件时,需要顾及到它周围的图件,例如元件不能重叠,网络不可短路,电源网络与其他信号线的间距应足够大等。这些要求称为 PCB 设计规划。大多数设计软件都提供一种功能,可以对设计完的 PCB 自动地进行设计规划检查,并给出详细的违规报告。设计者可根据违规报告进行修改。

### (7) PCB 仿真分析

PCB 仿真分析可使用所用软件自带功能,也可使用其他专用仿真软件。它能保证在物理制作之前,对 PCB 的信号处理进行仿真分析,以便进一步完善、修改。它同设计规划检查的内容是不同的。它主要分析布局布线对各参数的影响。

### (8) 存档输出

将设计好的印制板图保存为 PCB 图或其他类型的文档,以便今后使用、加工。如需要,可利用各种图形输出设备输出,如打印机、绘图仪等。

## 3. 多层 PCB 设计的注意事项(布线的原则)

在多层 PCB 布线时应注意以下事项:

- 高频信号线一定要短,不可以有尖角(90°直角),两根线之间的距离不宜平行、过近,否则可能会产生寄生电容。
- 如果是两面板,一面的线布成横线,一面的线布成竖线。尽量不要布成斜线。

- 如果使用自动布线无法完成所有布线,建议设计者首先手工将比较复杂的线布好,将布好的线锁定后,再使用自动布线功能,一般就可以完成全部布线。
- 一般来说,线宽一般为 0.3mm,间隔也为 0.3mm,这个长度约为 8~10mil。但是电源线、或者大电流线应该有足够宽度,一般需要 60~80mil。焊盘一般应为 64mil。如果是单面板,必须考虑焊盘,否则一般来说生产单面板的工艺都很差,所以单面板的焊盘尽量做得大一些,线要尽量粗一些。
- 做好屏蔽。铜膜线的地线应该在电路板的周边,同时将电路板上可以利用的空间全部使用铜箔做地线,增强屏蔽能力,并且防止寄生电容。多层板因为内层做为电源层和地线层,一般不会有屏蔽的问题。大面积敷铜应改用网格状,以防止焊接时板子产生气泡和因为热应力作用而弯曲。
- 焊盘的内孔尺寸如表 2-20 所示,必须从元件引线直径、公差尺寸、镀层厚度、孔径公差及孔金属化电镀层厚度等方面考虑,通常情况下以金属引脚直径加上 0.2mm 作为焊盘的内孔直径。例如,电阻的金属引脚直径为 0.5mm,则焊盘孔直径为 0.7mm,而焊盘外径应该为焊盘孔径加 1.2mm,最小应该为焊盘孔径加 1.0mm。当焊盘直径为 1.5mm 时,为了增加焊盘的抗剥离强度,可采用方形焊盘。对于孔直径小于 0.4mm 的焊盘,焊盘外径/焊盘孔直径为 0.5~3mm。对于孔直径 2mm 的焊盘,焊盘外径/焊盘孔直径为 1.5~2mm。焊盘一般应该补成泪滴状,这样线与焊盘的连接强度会大大增强。

表 2-20 常用的焊盘尺寸

焊盘孔直径/mm	焊盘外径/mm	焊盘孔直径/mm	焊盘外径/mm
0.4	1.5	1.0	2.5
0.5	1.5	1.2	3.0
0.6	2.0	1.6	3.5
0.8	2.0	2.0	4

- 地线的共阻抗干扰。电路图上的地线表示电路中的零电位,并用作电路中其他各点的公共参考点,在实际电路中由于地线(铜膜线)阻抗的存在,必然会带来共阻抗干扰,因此在布线时,不能将具有地线符号的点随便连接在一起,这可能引起有害的耦合而影响电路的正常工作。

#### 4. PCB 设计中的可靠性知识

目前电子器材用于各类电子设备和系统仍然以 PCB 为主要装配方式。实践证明,即使电路原理图设计正确,PCB 设计不当,也会对电子设备的可靠性产生不利影响。例如,如果 PCB 两条细平行线靠得很近,则会形成信号波形的延迟,在传输线的终端形成反射噪声。

因此,在设计 PCB 的时候,应注意采用正确的方法。

### (1) 地线设计

在电子设备中,接地是控制干扰的重要方法。如能将接地和屏蔽正确结合起来使用,可解决大部分干扰问题。电子设备中地线结构大致有系统地、机壳地(屏蔽地)、数字地(逻辑地)和模拟地等。在地线设计中应注意以下几点:

#### ① 正确选择单点接地与多点接地

在低频电路中,信号的工作频率小于 1MHz,它的布线和元件间的电感影响较小,而接地电路形成的环流对干扰影响较大,因而应采用一点接地。当信号工作频率大于 10MHz 时,地线阻抗变得很大,此时应尽量降低地线阻抗,应采用就近多点接地。当工作频率在 1~10MHz 时,如果采用一点接地,其地线长度不应超过波长的 1/20,否则应采用多点接地法。

#### ② 将数字电路与模拟电路分开

电路板上既有高速逻辑电路,又有线性电路,应使它们尽量分开,而两者的地线不要相混,分别与电源端地线相连。要尽量加大线性电路的接地面积。

#### ③ 尽量加粗接地线

若接地线很细,接地电位则随电流的变化而变化,致使电子设备的定时信号电平不稳,抗噪声性能变坏。因此应将接地线尽量加粗,使它能通过三倍于 PCB 的允许电流。如有可能,接地线的宽度应大于 3mm。

#### ④ 将接地线构成闭环路

设计只由数字电路组成的 PCB 的地线系统时,将接地线做成闭环路可以明显提高抗噪声能力。其原因在于:PCB 上有很多集成电路元件,尤其遇有耗电多的元件时,因受接地线粗细的限制,会在接地结构上产生较大的电位差,引起抗噪声能力下降,若将接地结构成环路,则会缩小电位差值,提高电子设备的抗噪声能力。

### (2) 电磁兼容性设计

电磁兼容性是指电子设备在各种电磁环境中仍能够协调、有效地进行工作的能力。电磁兼容性设计的目的是使电子设备既能抑制各种外来的干扰,使电子设备在特定的电磁环境中能够正常工作,又能减少电子设备本身对其他电子设备的电磁干扰。

#### ① 选择合理的导线宽度

由于瞬变电流在印制线条上所产生的冲击干扰主要是由印制导线的电感成分造成的,因此应尽量减小印制导线的电感量。印制导线的电感量与其长度成正比,与其宽度成反比,因而短而精的导线对抑制干扰是有利的。时钟引线、行驱动器或总线驱动器的信号线常常载有大的瞬变电流,印制导线要尽可能地短。对于分立元件电路,印制导线宽度在 1.5mm

左右时,即可完全满足要求;对于集成电路,印制导线宽度可在 $0.2\sim 1.0\text{mm}$ 之间选择。

### ② 采用正确的布线策略

采用平行走线可以减少导线电感,但导线之间的互感和分布电容增加,如果布局允许,最好采用井字形网状布线结构,具体做法是 PCB 的一面横向布线,另一面纵向布线,然后在交叉孔处用金属化孔相连。

为了抑制 PCB 导线之间的串扰,在设计布线时应尽量避免长距离的平行走线,尽可能拉开线与线之间的距离,信号线与地线及电源线尽可能不交叉。在一些对干扰十分敏感的信号线之间设置一根接地的印制线,可以有效地抑制串扰。

为了避免高频信号通过印制导线时产生的电磁辐射,在 PCB 布线时,还应注意以下几点:

- 尽量减少印制导线的不连续性,例如导线宽度不要突变,导线的拐角应大于 $90^\circ$ ,禁止环状走线等。
- 时钟信号引线最容易产生电磁辐射干扰,走线时应与地线回路相靠近,驱动器应紧挨着连接器。
- 总线驱动器应紧挨其欲驱动的总线。对于那些离开 PCB 的引线,驱动器应紧紧挨着连接器。
- 数据总线的布线应每两根信号线之间夹一根信号地线。最好是紧紧挨着最不重要的地址引线放置地回路,因为后者常载有高频电流。
- 在 PCB 布置高速、中速和低速逻辑电路时,应按照图 2-64 的方式排列元件。

### ③ 抑制反射干扰

为了抑制出现在印制线条终端的反射干扰,除了特殊需要之外,应尽可能缩短印制线的长度和采用慢速电路。必要时可加终端匹配,即在传输线的末端对地和电源端各加接一个相同阻值的匹配电阻。根据经验,对一般速度较快的 TTL 电路,其印制线条长于 $10\text{cm}$ 以上时就应采用终端匹配措施。匹配电阻的阻值应根据集成电路的输出驱动电流及吸收电流的最大值来决定。

### (3) 去耦电容配置

在直流电源回路中,负载的变化会引起电源噪声。例如在数字电路中,当电路从一个状态转换为另一种状态时,就会在电源线上产生一个很大的尖峰电流,形成瞬变的噪声电压。配置去耦电容可以抑制因负载变化而产生的噪声,是印制电路板的可靠性设计的一种常规做法,配置原则如下:

- 电源输入端跨接一个 $10\sim 100\mu\text{F}$ 的电解电容器,如果 PCB 的位置允许,采用 $100\mu\text{F}$

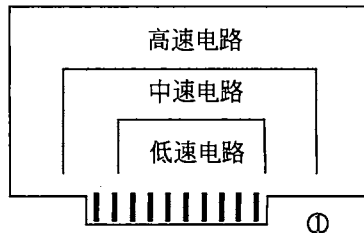


图 2-64 元件的排列方式

以上的电解电容器的抗干扰效果会更好。

- 为每个集成电路芯片配置一个  $0.01\mu\text{F}$  的陶瓷电容器。如遇到 PCB 空间小而装不下时,可每 4~10 个芯片配置一个  $1\sim 10\mu\text{F}$  钽电解电容器,这种元件的高频阻抗特别小,在  $500\text{kHz}\sim 20\text{MHz}$  范围内阻抗小于  $1\Omega$ ,而且漏电流很小 ( $0.5\mu\text{A}$  以下)。
- 对于噪声能力弱、关断时电流变化大的器件和 ROM、RAM 等存储型器件,应在芯片的电源线 ( $V_{\text{cc}}$ ) 和地线 (GND) 间直接接入去耦电容。
- 去耦电容的引线不能过长,特别是高频旁路电容不能带引线。

#### (4) PCB 的尺寸与器件的布置

PCB 大小要适中,过大时印制线条长,阻抗增加,不仅抗噪声能力下降,成本也高;过小,则散热不好,同时易受临近线条干扰。

在元件布置方面与其他逻辑电路一样,应把相互有关的元件尽量放得靠近些,这样可以获得较好的抗噪声效果,如图 2-65 所示。时钟发生器、晶振和 CPU 的时钟输入端都易产生噪声,要相互靠近些。易产生噪声的元件、小电流电路、大电流电路等应尽量远离逻辑电路,如有可能,应另做 PCB,这一点十分重要。

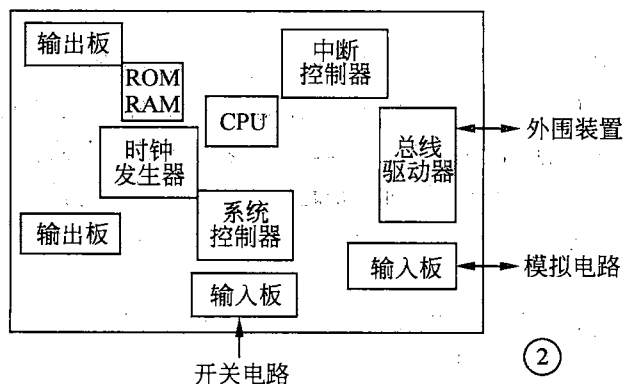


图 2-65 元件的布置

#### (5) 散热设计

从有利于散热的角度出发,PCB 最好是直立安装,板与板之间的距离一般不应小于 2cm,而且器件在 PCB 上的排列方式应遵循一定的规则:

- 对于采用自由对流空气冷却的设备,最好是将集成电路(或其他元件)按纵长方式排列,如图 2-66 所示;对于采用强制空气冷却的设备,最好是将集成电路(或其他元件)按横长方式排列,如图 2-67 所示。

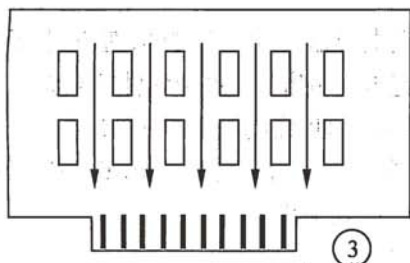


图 2-66 纵长方式排列

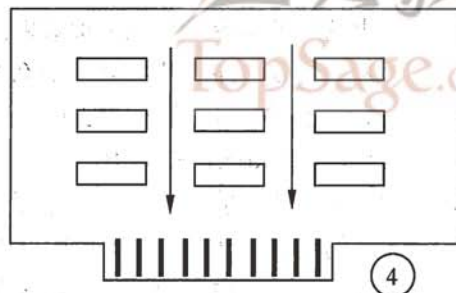


图 2-67 横长方式排列

- 同一块 PCB 上的元件应尽可能按其发热量大小及散热程度分区排列, 发热量小或耐热性差的元件 (如小信号晶体管、小规模集成电路、电解电容等) 放在冷却气流的最上游 (入口处), 发热量大或耐热性好的元件 (如功率晶体管、大规模集成电路等) 放在冷却气流最下游。
- 在水平方向上, 大功率元件尽量靠近 PCB 边沿布置, 以便缩短传热路径; 在垂直方向上, 大功率元件尽量靠近 PCB 上方布置, 以便减少这些元件工作时对其他元件温度的影响。
- 对温度比较敏感的元件最好安置在温度最低的区域 (如设备的底部), 千万不要将它放在发热器件的正上方, 多个元件最好是在水平面上交错布局。
- 设备内 PCB 的散热主要依靠空气流动, 所以在设计时要研究空气流动路径, 合理配置器件或 PCB。空气流动时总是趋向于阻力小的地方流动, 所以在 PCB 上配置元件时, 要避免在某个区域留有较大的空域。整机中多块 PCB 的配置也应注意同样的问题。

大量实践经验表明, 采用合理的器件排列方式, 可以有效地降低印制电路的温升, 从而使元件及设备的故障率明显下降。

以上所述只是 PCB 可靠性设计的一些通用原则, PCB 可靠性与具体电路有着密切的关系, 在设计中还需根据具体电路进行相应处理, 才能最大限度地保证 PCB 的可靠性。

### 2.7.3 电子设计

#### 1. 电子设计原理

随着半导体工艺水平的不断提高, 芯片中已经能够集成几百万个门电路, 一个完整的数字电子系统可以集成于一块芯片上 (System-On-a-Chip), 而传统产品设计需要经过人工设计、制作试验板、调试再修改多次循环才能定型, 完成这样的 SOC 设计已经十分困难。随着电子技术和计算机技术的飞速发展, 计算机应用水平的不断提高, 以及人们利用计算



机进行电子系统辅助设计,从而多大大提高了设计效率,减轻了设计人员的工作强度,缩短了设计周期,提高了设计成功率,减少了设计缺陷。上述这些优点,使电子设计自动化(Electronic Design Automation, EDA)成为现代电子设计的主流。

EDA 是指以计算机为工作平台,融合了应用电子技术、计算机技术、智能化技术最新成果而研制成的电子 CAD 通用软件包。利用 EDA 工具,电子工程师可以将电子产品的由电路设计、性能分析到 IC 设计图或 PCB 设计图整个过程在计算机上自动处理完成。

### 1) EDA 技术的发展历程

回顾近 30 年电子设计技术的发展历程,可将 EDA 技术分为 3 个阶段。

(1) 20 世纪 70 年代为 CAD 阶段,这一阶段人们开始用计算机辅助进行 IC 图编辑和 PCB 布局布线,取代了手工操作,产生了计算机辅助设计的概念。

(2) 20 世纪 80 年代为 CAE 阶段,与 CAD 相比,除了纯粹的图形绘制功能外,又增加了电路功能设计和结构设计,并且通过电气连接网络表将两者结合在一起,以实现工程设计,这就是计算机辅助工程的概念。CAE 的主要功能是:原理图输入,逻辑仿真,电路分析,自动布局布线,PCB 后分析。

(3) 20 世纪 90 年代为 ESDA 阶段。尽管 CAD/CAE 技术取得了巨大的成功,但并没有把人们从繁重的设计工作中彻底解放出来。在整个设计过程中,自动化和智能化程度还不高,各种 EDA 软件界面千差万别,学习使用困难,并且互不兼容,直接影响到设计环节的衔接。基于以上不足,人们开始追求贯彻整个设计过程的自动化,这就是电子系统设计自动化(Electronic System Design Automation, ESDA)。

### 2) ESDA 技术的基本特征

ESDA 代表了当今电子设计技术的最新发展方向,它的基本特征是:设计人员按照“自顶向下”的设计方法,对整个系统进行方案设计和功能划分,系统的关键电路用一片或几片专用集成电路(ASIC)实现,然后采用硬件描述语言(HDL)完成系统行为级设计,最后通过综合器和适配器生成最终的目标器件。下面介绍与 ESDA 基本特征有关的几个概念。

#### (1) “自顶向下”的设计方法

10 年前,电子设计的基本思路还是选择标准集成电路“自底向上”(Bottom-Up)的构造出一个新的系统,这样的设计方法就如同一砖一瓦建造金字塔,不仅效率低、成本高而且容易出错。

高层次设计给提供了一种“自顶向下”(Top-Down)的全新设计方法。这种设计方法首先从系统设计入手,在顶层进行功能方框图的划分和结构设计。在方框图一级进行仿真和纠错,并用硬件描述语言对高层次的系统行为进行描述,在系统一级进行验证。然后用综合优化工具生成具体门电路的网表,其对应的物理实现级可以是 PCB 或专用集成电路。由于设计的主要仿真和调试过程是在高层次上完成的,这一方面有利于早期发现结构设计



上的错误,避免设计工作的浪费,同时也减少了逻辑功能仿真的工作量,提高了设计的一次成功率。

## (2) ASIC 设计

现代电子产品的复杂度日益加深,一个电子系统可能由数万个中小规模集成电路构成,这就带来了体积大、功耗大、可靠性差的问题,解决这一问题的有效方法就是采用 ASIC 芯片进行设计。ASIC 按照设计方法的不同可分为全定制 ASIC、半定制 ASIC 和可编程 ASIC (也称为可编程逻辑器件)。设计全定制 ASIC 芯片时,设计师要定义芯片上所有晶体管的几何图形和工艺规则,最后将设计结果交由 IC 厂家掩膜制造完成。优点是:芯片可以获得最优的性能,即面积利用率高、速度快、功耗低。缺点是:开发周期长,费用高,只适合大批量产品开发。

半定制 ASIC 芯片的版图设计方法有所不同,分为门阵列设计法和标准单元设计法,这两种方法都是约束性的设计方法,其主要目的就是简化设计,以牺牲芯片性能为代价来缩短开发时间。

可编程逻辑芯片与上述掩膜 ASIC 的不同之处在于:设计人员完成版图设计后,在实验室内就可以烧制出自己的芯片,无需 IC 厂家的参与,大大缩短了开发周期。

可编程逻辑元件自 20 世纪 70 年代以来,经历了 PAL、GAL、CPLD、FPGA 几个发展阶段,其中 CPLD/FPGA 属高密度可编程逻辑元件,目前集成度已高达 200 万门/片,它将掩膜 ASIC 集成度高的优点和可编程逻辑元件设计生产方便的特点结合在一起,特别适合于样品研制或小批量产品开发,使产品能以最快的速度上市,而当市场扩大时,它可以很容易的转由掩膜 ASIC 实现,因此开发风险也大为降低。

上述 ASIC 芯片,尤其是 CPLD/FPGA 元件,已成为现代高层次电子设计方法的实现载体。

## (3) 硬件描述语言

硬件描述语言(Hardware Description Language, HDL)是一种用于设计硬件电子系统的计算机语言。它用软件编程的方式来描述电子系统的逻辑功能、电路结构和连接形式,与传统的门级描述方式相比,它更适合大规模系统的设计。例如一个 32 位的加法器,利用图形输入软件需要输入 500~1000 个门,而利用 HDL 语言只需要书写一行  $A=B+C$  即可,而且 HDL 语言可读性强,易于修改和发现错误。早期的硬件描述语言,如 ABEL-HDL、AHDL,由不同的 EDA 厂商开发,互不兼容,而且不支持多层次设计,层次间翻译工作要由人工完成。为了克服以上不足,1985 年美国国防部正式推出了 VHDL (Very High Speed IC Hardware Description Language)语言,1987 年 IEEE 采纳 VHDL 为硬件描述语言标准(IEEE STD-1076)。

VHDL 是一种全方位的硬件描述语言,包括系统行为级、寄存器传输级和逻辑门级多

个设计层次,支持结构、数据流、行为 3 种描述形式的混合描述,因此 VHDL 几乎覆盖了以往各种硬件描述语言的功能,整个自顶向下或自底向上的电路设计过程都可以用 VHDL 来完成。VHDL 还具有以下优点:

- VHDL 的宽范围描述能力使它成为高层次设计的核心,将设计人员的工作重心提高到了系统功能的实现与调试,而花较少的精力于物理实现。
- VHDL 可以用简洁明确的代码描述来进行复杂控制逻辑的设计,灵活且方便,而且也便于设计结果的交流、保存和重用。
- VHDL 的设计不依赖于特定的元件,方便了工艺的转换。
- VHDL 是一个标准语言,为众多的 EDA 厂商支持,因此移植性好。

#### (4) 系统框架结构

EDA 系统框架结构(Framework)是一套配置和使用 EDA 软件包的规范,目前主要的 EDA 系统都建立了框架结构,如 Cadence 公司的 Design Framework, Mentor 公司的 Falcon Framework,而且这些框架结构都遵守国际 CFI 组织(CAD Framework Initiative)制定的统一技术标准。Framework 能将来自不同 EDA 厂商的工具软件进行优化组合,集成在一个易于管理的统一的环境之下,而且还支持任务之间、设计师之间以及整个产品开发过程中信息的传输与共享,是并行工程和“自顶向下”设计方法的实现基础。

### 3) EDA 技术的基本设计方法

#### (1) 电路级设计

电子工程师接受系统设计任务后,首先确定设计方案,同时要选择能实现该方案的合适元件,然后根据具体的元件设计电路原理图。接着进行第一次仿真,包括数字电路的逻辑模拟、故障分析,模拟电路的交直流分析、瞬态分析。系统在进行仿真时,必须要有元件模型库的支持,计算机上模拟的输入输出波形代替了实际电路调试中的信号源和示波器。这一次仿真主要是检验设计方案在功能方面的正确性。

仿真通过后,根据原理图产生的电气连接网络表进行 PCB 的自动布局布线。在制作 PCB 之前还可以进行后分析,包括热分析、噪声及窜扰分析、电磁兼容分析、可靠性分析等,并且可以将分析后的结果参数反标回电路图,进行第二次仿真,也称为后仿真,这一次仿真主要是检验 PCB 在实际工作环境中的可行性。

由此可见,电路级的 EDA 技术使电子工程师在实际的电子系统产生前,就可以全面了解系统的功能特性和物理特性,从而将开发风险消灭在设计阶段,缩短了开发时间,降低了开发成本。

#### (2) 系统级设计

进入 20 世纪 90 年代以来,电子信息类产品的开发明显出现两个特点:一是产品的复杂程度增加;二是产品的上市时限紧迫。然而电路级设计本质上是基于门级描述的单层次

设计,设计的所有工作(包括设计输入,仿真和分析,设计修改等)都是在基本逻辑门这一层次上进行的,显然这种设计方法不能适应新的形势,为此引入了一种高层次的电子设计方法,也称为系统级的设计方法。

高层次设计是一种“概念驱动式”设计,设计人员无须通过门级原理图描述电路,而是针对设计目标进行功能描述,由于摆脱了电路细节的束缚,设计人员可以把精力集中于创造性的方案与概念构思上,一旦这些概念构思以高层次描述的形式输入计算机后,EDA系统就能以规则驱动的方式自动完成整个设计。这样,新的概念得以迅速有效地成为产品,大大缩短了产品的研制周期。不仅如此,高层次设计只是定义系统的行为特性,可以不涉及实现工艺,在厂家综合库的支持下,利用综合优化工具可以将高层次描述转换成针对某种工艺优化的网表,工艺转化变得轻松容易。

高层次设计步骤如下:

- ① 按照“自顶向下”的设计方法进行系统划分。
  - ② 输入 VHDL 代码,这是高层次设计中最为普遍的输入方式。此外,还可以采用图形输入方式(框图、状态图等),这种输入方式具有直观、容易理解的优点。
  - ③ 将以上的设计输入编译成标准的 VHDL 文件。对于大型设计,还要进行代码级的功能仿真,主要是检验系统功能设计的正确性,因为对于大型设计,综合、适配要花费数小时,在综合前对源代码仿真,就可以大大减少设计重复的次数和时间,一般情况下,可略去这一仿真步骤。
  - ④ 利用综合器对 VHDL 源代码进行综合优化处理,生成门级描述的网络表文件,这是将高层次描述转化硬件电路的关键步骤。综合优化是针对 ASIC 芯片供应商的某一产品系列进行的,所以综合的过程要在相应的厂家综合库支持下才能完成。综合后,可利用产生的网表文件进行适配前的时序仿真,仿真过程不涉及具体元件的硬件特性,是较为粗略的一般设计,这一仿真步骤也可略去。
  - ⑤ 利用适配器将综合后的网络表文件针对某一具体的目标元件进行逻辑映射操作,包括底层元件配置、逻辑分割、逻辑优化、布局布线。适配完成后,产生多项设计结果:
    - 适配报告,包括芯片内部资源利用情况,设计的布尔方程描述情况等。
    - 适配后的仿真模型。
    - 元件编程文件。
- 根据适配后的仿真模型,可以进行适配后的时序仿真,因为已经得到元件的实际硬件特性(如时延特性),所以仿真结果能比较精确的预期未来芯片的实际性能。如果仿真结果达不到设计要求,就需要修改 VHDL 源代码或选择不同速度品质的器件,直至满足设计要求。
- ⑥ 将适配器产生的元件编程文件通过编程器或下载电缆载入到目标芯片 FPGA 或

CPLD 中。如果是大批量产品开发,通过更换相应的厂家综合库,可以很容易转由 ASIC 形式实现。

## 2.7.4 电子电路测试

### 1. 电子电路测试原理与方法

测试的意义在于检查设计的具体电路是否能够像设计者要求的那样正确工作。测试的任务就是确认 IC 芯片内部有没有隐藏的故障。由于半导体工艺水平的提高,芯片的复杂程度增加,测试难度越来越大。现在已经不能直接从 I/O 接口来完整地控制和观察芯片和内部节点的电学行为。

判断故障是否存在即只判断有无故障,称为故障检测 (Fault Detection); 若不仅判断故障是否存在,而且指出故障位置,称为故障诊断 (Fault Diagnosis) 或故障定位 (Fault Isolation)。测试与仿真(模拟)是不一样的。仿真是对设计过程中得到的电路参数验证其正确性,是 IC 芯片在产品未生产之前进行的;测试是判断产品是否合格,是在 IC 芯片生产出来之后进行的,是产品制造过程中的工序之一。

经典的测试方法是用一个测试设备进行的。将被测 IC 芯片放到测试仪器上,测试设备根据需要产生一系列测试输入信号,加到被测元件上,在被测元件输出端得到输出信号,测试设备将实际输出信号和预期输出信号比较,如果吻合,表示测试通过,否则不能通过。

可测试设计的 3 个方面是测试生成、测试验证、测试设计。测试生成是指产生验证 IC 芯片行为的一组测试码。测试验证指给定测试集合的有效性测度,这通常是通过故障仿真来估算的。测试设计是设计者在电路设计阶段就考虑芯片的测试结构问题,在设计用户逻辑的同时,还要设计测试电路。

现代电子设计方法包含可测试设计。内建测试系统 (BIST) 是 SOC 片上系统的重要结构之一。JTAG 测试接口是 IC 芯片测试方法的标准。

芯片测试分为样品测试和产品测试。样品测试需要非常严格的测试矢量,并检查可能导致芯片功能错误的故障。生产测试主要注意测试成本,给出 IC 芯片功能检查通过或不通过的信号。

可测试设计作为 EDA 工程的重要组成部分,受到设计者、生产者重视。他们从不同的角度提出了许多改善可测试性的建议:

- 将复杂的电路分块,使模块测试更加容易。
- 采用电路技术,使测试矢量生成难度减少。
- 改进可控性和可观察性。
- 添加自检测电路。



## 2. 硬件抗干扰测试

影响系统可靠安全运行的主要因素主要来自系统内部和外部的各种电气干扰，并受系统结构设计、元件选择、安装、制造工艺影响。这些都构成嵌入式系统的干扰因素，常会导致嵌入式系统运行失常，轻则影响产品质量和产量，重则会导致事故，造成重大经济损失。

形成干扰的基本要素有 3 个：

- 干扰源。指产生干扰的元件、设备或信号，用数学语言描述如下： $du/dt$ 。  $du/dt$  大的地方就是干扰源，如雷电、继电器、可控硅、电机、高频时钟等都可能成为干扰源。
- 传播路径。指干扰从干扰源传播到敏感器件的通路或媒介。典型的干扰传播路径是通过导线的传导和空间的辐射。
- 敏感器件。指容易被干扰的对象，如 A/D 转换器、D/A 转换器、单片机、数字 IC、弱信号放大器等。

### 1) 干扰的分类

#### (1) 干扰的分类

干扰的分类有好多种，通常可以按照噪声产生的原因、传导方式、波形特性等进行不同的分类。按产生的原因，可分为放电噪声、高频振荡噪声、浪涌噪声；按传导方式，可分为共模噪声和串模噪声；按波形分，可分为持续正弦波、脉冲电压、脉冲序列等。

#### (2) 干扰的耦合方式

干扰源产生的干扰信号是通过一定的耦合通道才对系统产生作用的。因此，有必要看看干扰源和被干扰对象之间的传递方式。干扰的耦合方式，无非是通过导线、空间、公共线等，细分下来，主要有以下几种。

- 直接耦合：这是最直接的方式，也是系统中存在最普遍的一种方式。比如干扰信号通过电源线侵入系统。对于这种形式，最有效的方法就是加入去耦电路。
- 公共阻抗耦合：这也是常见的耦合方式，这种形式常常发生在两个电路电流有共同通路的情况。为了防止这种耦合，通常在电路设计上就要考虑，使干扰源和被干扰对象间没有公共阻抗。
- 电容耦合：又称电场耦合或静电耦合，是由于分布电容的存在而产生的耦合。
- 电磁感应耦合：又称磁场耦合，是由于分布电磁感应而产生的耦合。
- 漏电耦合：这种耦合是纯电阻性的，在绝缘不好时就会发生。

### 2) 常用硬件抗干扰技术

针对形成干扰的 3 要素，采取的抗干扰主要有以下手段。

#### (1) 抑制干扰源

抑制干扰源就是尽可能的减小干扰源的  $du/dt$  和  $di/dt$ 。这是抗干扰设计中最优先考虑和最重要的原则，常常会起到事半功倍的效果。减小干扰源的  $du/dt$  主要是通过在干扰源两端并联电容来实现。减小干扰源的  $di/dt$  则是在干扰源回路串联电感或电阻及增加续流二极管来实现。

抑制干扰源的常用措施如下：

- 继电器线圈增加续流二极管，消除断开线圈时产生的反电动势干扰。仅加续流二极管会使继电器的断开时间滞后，增加稳压二极管后继电器在单位时间内可动作更多的次数。
- 在继电器接点两端并接火花抑制电路（一般是 RC 串联电路，电阻一般选几 KB 到几十 KB，电容选  $0.01\mu\text{F}$ ），减小电火花影响。
- 给电机加滤波电路，注意电容、电感引线要尽量短。
- 电路板上每个 IC 要并接一个  $0.01\mu\text{F} \sim 0.1\mu\text{F}$  高频电容，以减小 IC 对电源的影响。注意高频电容的布线，连线应靠近电源端并尽量粗短，否则，等于增大了电容的等效串联电阻，会影响滤波效果。
- 布线时避免  $90^\circ$  折线，减少高频噪声发射。
- 可控硅两端并接 RC 抑制电路，减小可控硅产生的噪声（这个噪声严重时可能会把可控硅击穿的）。

## （2）切断干扰传播路径

按干扰的传播路径可分为传导干扰和辐射干扰两类。

所谓传导干扰是指通过导线传播到敏感元件的干扰。高频干扰噪声和有用信号的频带不同，可以通过在导线上增加滤波器的方法切断高频干扰噪声的传播，有时也可加隔离光耦来解决。电源噪声的危害最大，要特别注意处理。

所谓辐射干扰是指通过空间辐射传播到敏感元件的干扰。一般的解决方法是增加干扰源与敏感元件的距离，用地线把它们隔离和在敏感元件上加屏蔽罩。

切断干扰传播路径的常用措施如下：

- 充分考虑电源对嵌入式系统的的影响。电源做得好，整个电路的抗干扰就解决了一大半。许多嵌入式系统对电源噪声很敏感，要给嵌入式系统电源加滤波电路或稳压器，以减小电源噪声对嵌入式系统的干扰。比如，可以利用磁珠和电容组成  $\pi$  形滤波电路，当然条件要求不高时也可用  $100\Omega$  电阻代替磁珠。
- 如果嵌入式微处理器的 I/O 接口用来控制电机等噪声器件，在 I/O 接口与噪声源之间应加隔离（增加  $\pi$  形滤波电路）。
- 注意晶振布线。晶振与嵌入式微处理器引脚尽量靠近，用地线把时钟区隔离起来，晶振外壳接地并固定。

- 电路板合理分区，如强、弱信号，数字、模拟信号。尽可能把干扰源（如电机、继电器）与敏感元件（如单片机）远离。
- 用地线把数字区与模拟区隔离。数字地与模拟地要分离，最后在一点接于电源地。A/D、D/A 芯片布线也以此为原则。
- 嵌入式微处理器和大功率器件的地线要单独接地，以减小相互干扰。大功率器件尽可能放在电路板边缘。
- 在嵌入式微处理器的 I/O 接口、电源线、电路板连接线等关键地方使用抗干扰元件如磁珠、磁环、电源滤波器、屏蔽罩，可显著提高电路的抗干扰性能。

### 3) 提高敏感元件的抗干扰性能

提高敏感元件的抗干扰性能是指从敏感元件这边考虑尽量减少对干扰噪声的拾取，以及从不正常状态尽快恢复的方法。

提高敏感元件抗干扰性能的常用措施如下：

- 布线时尽量减少回路环的面积，以降低感应噪声。
- 布线时，电源线和地线要尽量粗。除减小压降外，更重要的是降低耦合噪声。
- 对于嵌入式微处理器闲置的 I/O 接口，不要悬空，要接地或接电源。其他 IC 的闲置端在不改变系统逻辑的情况下接地或接电源。
- 对嵌入式微处理器使用电源监控及看门狗电路，如 IMP809、IMP706、IMP813、X5043 和 X5045 等，可大幅度提高整个电路的抗干扰性能。
- 在速度能满足要求的前提下，尽量降低嵌入式微处理器的晶振和选用低速数字电路。
- IC 器件尽量直接焊在电路板上，少用 IC 座。

### 4) 其他常用抗干扰措施

- 交流端用电感电容滤波：去掉高频低频干扰脉冲。
- 变压器双隔离措施：变压器初级输入端串接电容，初、次级线圈间屏蔽层与初级间电容中心接点接大地，次级外屏蔽层接 PCB，这是硬件抗干扰的关键手段。次级加低通滤波器，以吸收变压器产生的浪涌电压。
- 采用集成式直流稳压电源：有过流、过压、过热等保护作用。
- I/O 接口采用光电、磁电、继电器隔离，同时去掉公共地。
- 通信线用双绞线，排除平行互感。
- 防雷用电光纤隔离最为有效。
- A/D 转换用隔离放大器或采用现场转换以减少误差。
- 外壳接大地以解决人身安全及防外界电磁场干扰。
- 加复位电压检测电路。防止复位不充分，CPU 就工作，尤其有 EEPROM 的元件，

复位不充分会改变 EEPROM 的内容。

- PCB 工艺抗干扰。
  - ◆ 电源线加粗，合理走线、接地，三总线分开以减少互感振荡。
  - ◆ CPU、RAM、ROM 等主芯片、VCC 和 GND 之间接电解电容及瓷片电容，去掉高、低频干扰信号。
  - ◆ 独立系统结构，减少接插件与连线，提高可靠性，减少故障率。
  - ◆ 集成块与插座接触可靠，用双簧插座，最好集成块直接焊在 PCB 上，防止元件接触不良故障。
  - ◆ 有条件地采用四层以上 PCB，中间两层为电源及地。



## 第3章 嵌入式系统软件及操作系统知识

### 3.1 嵌入式软件基础

#### 3.1.1 嵌入式软件概述

嵌入式软件是指应用在嵌入式计算机系统当中的各种软件。在嵌入式系统的发展初期,软件的种类很少,规模也很小,基本上都是硬件的附属品。随着嵌入式系统应用的发展,特别是随着后 PC 时代的来临,嵌入式软件的种类和规模都得到了极大的发展,形成了一个完整、独立的体系。但是作为嵌入式系统的一个组成部分,无论嵌入式软件如何发展,都摆脱不了整个系统对它的影响。因此,对于嵌入式软件而言,它除了具有通用软件的一般特性,同时还具有一些与嵌入式系统密切相关的特点。这些特点包括:

##### (1) 规模较小

由于嵌入式系统的资源一般比较有限,所以嵌入式软件必须尽可能地精简,才能适应这种状况,多数的嵌入式软件都在几 MB 以内。

##### (2) 开发难度大

与普通计算机上的软件不同,嵌入式软件的运行环境和开发环境比较复杂,这就加大了它的开发难度。首先,由于硬件资源有限,使得嵌入式软件在时间和空间上都受到严格的限制,要想开发出运行速度快、存储空间少、维护成本低的软件,不是一件容易的事情,需要开发人员对编程语言、编译器和操作系统有深刻的了解。其次,嵌入式软件一般都要涉及到底层软件的开发,应用软件的开发也是直接基于操作系统的,这就需要开发人员具有扎实的软、硬件基础,能灵活运用不同的开发手段和工具,具有较丰富的开发经验。最后,对于嵌入式软件来说,它的开发环境与运行环境是不同的。嵌入式软件是在目标系统上运行的,但开发工作却要在另外的开发系统中进行,当程序员将应用软件调试无误后,再把它放到目标系统上去。这与通常的软件开发过程是不同的。

##### (3) 实时性和可靠性要求高

实时性是嵌入式系统的一个重要特征,许多嵌入式系统要求具有实时处理的能力,这种实时性主要是靠软件层来体现的。软件对外部事件做出反应的时间必须要快,在某些情况下还要求是确定的、可重复实现的,不管系统当时的内部状态如何,都是可以预测的。

同时,对于事件的处理一定要在限定的时间期限之前完成,否则就有可能引起系统的崩溃。例如,火箭飞行控制系统就是实时的,它对飞行数据采集和燃料喷射时机的把握要求非常的准确,否则就难以达到精确控制的目的,从而导致飞行控制的失败。

与实时性相应的是可靠性,因为实时系统往往应用在一些比较重要的领域,如航天控制、核电站、工业机器人等,如果软件出了问题,那么后果是非常严重的,所以要求这种嵌入式软件的可靠性必须非常高。

#### (4) 要求固化存储

为了提高系统的启动速度、执行速度和可靠性,嵌入式系统中的软件一般都固化在存储器芯片或单片机本身中,而不是像通常的计算机系统那样,存储在磁盘等载体中。

### 3.1.2 嵌入式软件分类

按通常的分类方法,嵌入式软件可以分为三大类:系统软件、应用软件和支撑软件。

- 系统软件:控制和管理嵌入式系统资源,为嵌入式应用提供支持的各种软件,如设备驱动程序、嵌入式操作系统、嵌入式中间件等。
- 应用软件:嵌入式系统中的上层软件,它定义了嵌入式设备的主要功能和用途,并负责与用户进行交互。应用软件是嵌入式系统功能的体现,一般面向于特定的应用领域,如飞行控制软件、手机软件、MP3 播放软件、电子地图软件等。
- 支撑软件:辅助软件开发的工具软件,如系统分析设计工具、在线仿真工具、交叉编译器、源程序模拟器和配置管理工具等。

在嵌入式系统当中,由于它的硬件配置一般比较低,无法运行太多、太复杂的软件。因此,对于系统软件和应用软件来说,它们是运行在目标平台的,即嵌入式设备上。而对于各种软件开发工具来说,它们大部分都运行在开发平台上,这个开发平台一般是一台普通的 PC 机,运行 Windows 或 Linux 操作系统。本章主要讨论的是运行在嵌入式设备上的软件,即系统软件和应用软件。

### 3.1.3 嵌入式软件的体系结构

#### 1. 无操作系统的情形

在嵌入式系统的发展初期,由于硬件的配置比较低,而且系统的应用范围也比较有限,主要集中在控制领域,对于是否有系统软件的支持,要求还不是很强烈。因此在那个阶段,嵌入式软件的设计主要是以应用为核心,应用软件直接建立在硬件上,没有专门的操作系统,软件的规模也很小,基本上属于硬件的附属品。

在具体实现上,无操作系统的嵌入式软件主要有两种实现方式:循环轮转和前后台系统。

### (1) 循环轮转方式

如图 3-1 所示, 循环轮转方式的基本思路是: 把系统的功能分解为若干个不同的任务, 然后把它们包含在一个永不结束的循环语句当中, 按照顺序逐一执行。当执行完一轮循环后, 又回到循环体的开头重新执行。

循环轮转方式的优点是简单、直观、开销小、可预测。软件的开发就是一个典型的基于过程的程序设计问题, 可以按照自顶向下、逐步求精的方式, 将系统要完成的功能逐级划分成若干个小的功能模块, 像搭积木一样搭起来。由于整个系统只有一条执行流程和一个地址空间, 不需要任务之间的调度和切换, 所以系统的管理开销很少。而且由于程序的代码都是固定的, 函数之间的调用关系也是明确的, 所以整个系统的执行过程是可预测的, 这对于一些实时控制系统来说是非常重要的。

循环轮转方式的缺点是过于简单, 所有的代码都必须按部就班地顺序执行, 无法处理异步事件, 缺乏并行处理的能力。而在现实世界当中, 事件都是并行出现的, 而且有些事件比较紧急, 当它们发生的时候, 必须马上进行处理, 不能等到下一轮循环再来处理。另外, 这种方案没有硬件上的时间控制机制, 无法实现定时功能。

### (2) 前后台系统

前后台系统就是在循环轮转方式的基础上, 增加了中断处理功能。

如图 3-2 所示, 中断服务程序 (Interrupt Service Routine, ISR) 负责处理异步事件, 这部分可以看成是前台程序 (foreground)。而后台程序 (background) 一般是一个无限的循环, 负责掌管整个嵌入式系统软、硬件资源的分配、管理以及任务的调度, 是一个系统管理调度程序。一般情形下, 后台程序也叫任务级程序, 前台程序也叫事件处理级程序。在系统运行时, 后台程序会检查每个任务是否具备运行条件, 通过一定的调度算法来完成相应的操作。而对于实时性要求特别严格的操作通常由中断来完成。为了提高系统性能, 大多数的中断服务程序只做一些最基本的操作, 例如, 把来自于外设的数据复制到缓冲区、标记中断事件的发生等, 其余的事情会延迟到后台程序去完成, 这样就不会因为在中断服务程序中耽误太长的时间而影响后续和其他的中断。

实际上, 前后台系统的实时性比预计的要差。这是因为前后台系统认为所有的任务具有相同的优先级别, 即是平等的, 而且任务的执行又是通过先进先出的队列排队, 因而对那些实时性要求很高的任务不能立刻进行处理。但由于这类系统的结构比较简单, 几乎不需要 RAM/ROM 的额外开销, 因而在一些简单的嵌入式应用中被广泛使用, 如微波炉、电

```
for (;;)
{
    任务 1 的部分工作;
    任务 2 的部分工作;
    任务 3 的部分工作
}
```

图 3-1 循环轮转方式

话机、电子玩具等。

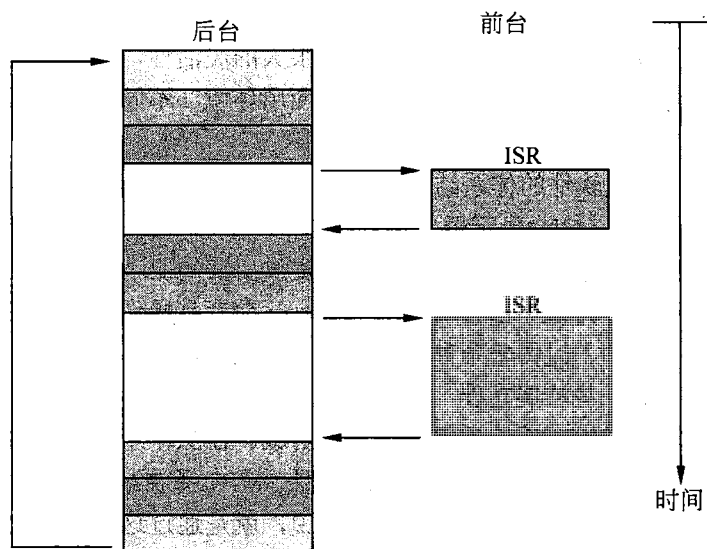


图 3-2 前后台系统

## 2. 有操作系统的情形

从 20 世纪 80 年代开始, 嵌入式软件进入了操作系统的阶段。这一阶段的标志是操作系统出现在嵌入式系统上, 程序员在开发应用程序的时候, 不是直接面对嵌入式硬件设备, 而是在操作系统的基础上编写, 嵌入式软件开发环境也得到了一定的应用。如今, 嵌入式操作系统在嵌入式应用中用得越来越广泛, 尤其是在功能复杂、系统庞大的应用中显得愈来愈重要。这种开发方式主要有以下三个优点:

### (1) 提高了系统的可靠性

在控制系统中, 出于安全方面的考虑, 要求系统起码不能崩溃, 而且还要有自愈能力, 这就需要在硬件设计和软件设计这两个方面来提高系统的可靠性和抗干扰性, 尽可能地减少安全漏洞和不可靠的隐患。以往的前后台系统在遇到强干扰时, 可能会使应用程序产生异常、出错, 甚至死循环的现象, 从而造成系统的崩溃。而嵌入式操作系统管理的系统, 这种干扰可能只会引起系统中的某一个进程被破坏, 这时可以通过系统的监控进程对其进行修复。

### (2) 提高了系统的开发效率, 降低了开发成本, 缩短了开发周期

在嵌入式操作系统环境下, 开发一个复杂的应用程序, 通常可以按照软件工程的思想, 将整个程序分解为多个任务模块, 每个任务模块的调试、修改几乎不影响其他模块。而且

商业软件一般都提供了良好的多任务调试环境，这样就大大提高了系统的开发效率。

(3) 有利于系统的扩展和移植

在嵌入式操作系统环境下开发应用程序具有很大的灵活性，操作系统本身可以剪裁，外设、相关应用也可以配置，软件可以在不同的应用环境、不同的处理器芯片之间移植，软件构件可复用。

图 3-3 所示是嵌入式软件的体系结构图。最底层是嵌入式硬件，包括嵌入式微处理器、存储器和键盘、输入笔、LCD 显示器等输入/输出设备。紧接在硬件层之上的，是设备驱动层，它负责与硬件直接打交道，并为上层软件提供所需的驱动支持。设备驱动层的上面是操作系统层，它可以分为基本部分和扩展部分——前者是操作系统的核心，负责整个系统的任务调度、存储管理、时钟管理和中断管理等功能，这一部分是基础和必备的；后者则是系统为用户提供的一些扩展功能，包括网络、文件系统、图形用户界面 GUI、数据库等，这一部分的内容可以根据系统的需要来进行剪裁。在操作系统的上面，是一些中间件软件，再上面就是各种应用软件了，如网络浏览器、MP3 播放器、文本编辑器、电子邮件客户端、电子游戏等。对于嵌入式系统的用户来说，就是通过这些应用软件来跟系统打交道的。

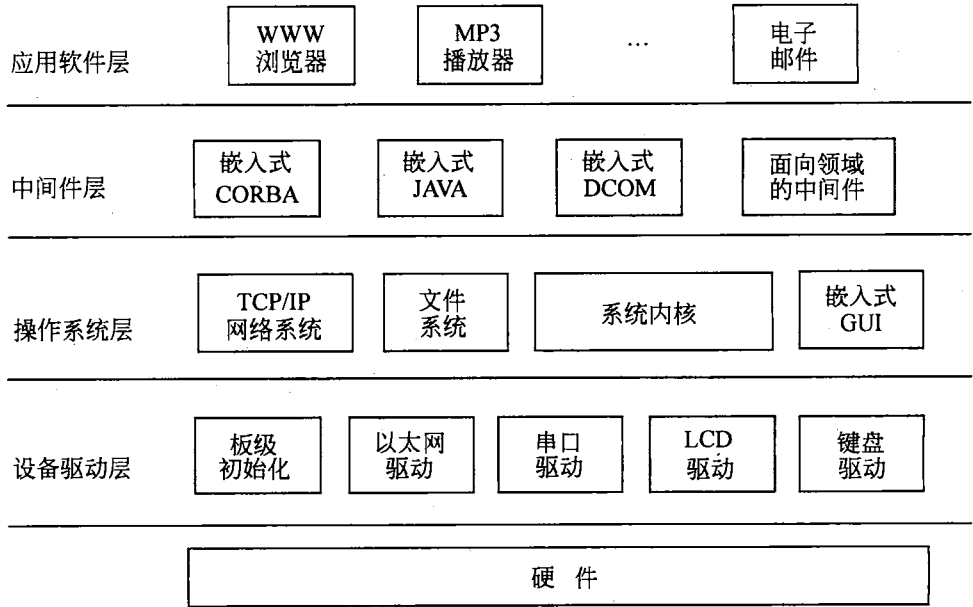


图 3-3 嵌入式软件体系结构

### 3.1.4 设备驱动层

大多数的嵌入式硬件设备都需要某种类型的软件初始化和管理工作。这部分工作是由设备驱动层来完成的，它负责直接与硬件打交道，对硬件进行管理和控制，并为上层软件提供所需的驱动支持。

#### 1. 板级支持包

设备驱动层也叫板级支持包（Board Support Package, BSP），它包含了嵌入式系统中所有与硬件相关的代码。BSP 的基本思想是把嵌入式操作系统与具体的硬件平台隔离开来，也就是说，在 BSP 当中，把所有与硬件相关的代码都封装起来，并向上提供一个虚拟的硬件平台，而操作系统就运行在这个虚拟的硬件平台上。它使用一组定义好的编程接口来与 BSP 进行交互，并通过 BSP 来访问真正的硬件。BSP 在嵌入式系统中的角色，类似于 PC 系统中的 BIOS 和驱动程序。

为什么在嵌入式系统中要引入 BSP 呢？我们知道，对于一个成熟的商用操作系统而言，为了在业界得到广泛的应用，就必须能够支持种类众多的硬件平台，并实现应用程序的硬件无关性。一般来说，这种无关性是由操作系统来实现的。但是对于嵌入式系统来说，它没有像 PC 机那样广泛使用的各种工业标准和统一的硬件结构。各种嵌入式系统的应用需求各不相同，这就决定了它们一般都选用专门定制的硬件环境——从核心的处理器到外部芯片，在硬件结构上都有很大的不同。这种变化众多的硬件环境就决定了无法完全由操作系统来实现上层软件与底层硬件之间的无关性。因此各种商用的嵌入式操作系统，都采用了分层设计的思想，将系统中与硬件直接相关的一层软件独立出来，称之为板级支持包。顾名思义，BSP 是针对某个特定的单板而设计的，如果某个单板没有相应的支持软件包，那么操作系统就不能在它上面运行。另外，BSP 对于用户（指系统开发人员）也是开放的，用户可以根据不同的硬件需求对其进行改动或二次开发。一般来说，针对某一类 CPU 的硬件单板，嵌入式操作系统都会提供相应的演示版本的 BSP，即所谓的最小系统 BSP。因此，在实际开发一个嵌入式系统的时候，人们通常会首先找到一个与自己的硬件系统相近的演示版本的 BSP，并以此为基础，进行修改和完善，以适应不同单板的需求。

对于不同的嵌入式操作系统，BSP 的具体结构和组成也各不相同。但一般来说，BSP 主要包括两个方面的内容：引导加载程序 BootLoader 和设备驱动程序。

#### 2. 引导加载程序

引导加载程序 BootLoader 是嵌入式系统加电后运行的第一段软件代码。我们知道，在桌面 PC 机当中，它的引导加载程序由两部分代码组成：位于只读存储器 ROM 中的 BIOS 和位于硬盘主引导记录（Master Boot Record, MBR）中的 BootLoader 引导程序（如 LILO 和 GRUB）。BIOS 在完成硬件检测和资源分配后，将硬盘 MBR 中的引导程序读到系统的

内存当中，然后将控制权交给它，由它负责把操作系统的内核映像从硬盘读入到内存，然后跳转到内核的入口点去运行，即启动操作系统。但是在嵌入式系统当中，通常没有像 BIOS 那样的固件程序，所以整个系统的加载启动任务就完全由 BootLoader 来完成。例如，在一个基于 ARM7TDMI 内核的嵌入式系统中，系统在上电或复位时一般从地址 0x00000000 处开始执行，而在这个地址处安排的通常就是系统的 BootLoader 程序。

那么到底什么是 BootLoader 呢？简单地说，BootLoader 就是在操作系统内核运行之前运行的一小段程序。通过这段程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境设置到一个合适的状态，以便为最终调用操作系统内核做好准备。

在嵌入式系统中，BootLoader 的实现高度依赖于具体的硬件平台，对于不同的 CPU 体系结构和板级设备配置，需要不同的 BootLoader。因此，要想建立一个通用的 BootLoader 几乎是不可能的。但是，一般来说，它主要包含以下的一些基本功能。

- 片级初始化：主要完成微处理器的初始化，包括设置微处理器的核心寄存器和控制寄存器、微处理器的核心工作模式及其局部总线模式等。片级初始化把微处理器从上电时的默认状态逐步设置成系统所要求的工作状态。这是一个纯硬件的初始化过程。
- 板级初始化：通过正确地设置各种寄存器的内容来完成微处理器以外的其他硬件设备的初始化。例如，初始化 LED 显示设备、初始化定时器、设置中断控制寄存器、初始化串口通信、初始化内存控制器、建立内存空间的地址映射等。在此过程中，除了要设置各种硬件寄存器以外，还要设置某些软件的数据结构和参数。因此，这是一个同时包含有软件和硬件在内的初始化过程。
- 加载内核：将操作系统和应用程序的映像从 Flash 存储器复制到系统的内存当中，然后跳转到系统内核的第一条指令处继续执行。

### 3. 设备驱动程序

如前所述，在一个嵌入式系统当中，操作系统是可能有也可能无的。但无论如何，设备驱动程序是必不可少的。所谓的设备驱动程序，就是一组库函数，用来对硬件进行初始化和管理的，并向上层软件提供良好的访问接口。

对于不同的硬件设备来说，它们的功能是不一样的，所以它们的设备驱动程序也是不一样的。但是一般来说，大多数的设备驱动程序都会具备以下的一些基本功能。

- 硬件启动：在开机上电或系统重启的时候，对硬件进行初始化。
- 硬件关闭：将硬件设置为关机状态。
- 硬件停用：暂停使用这个硬件。
- 硬件启用：重新启用这个硬件。
- 读操作：从硬件中读取数据。

- 写操作：往硬件中写入数据。

除了以上这些普遍适用的功能之外，设备驱动程序还可能有很多额外的、特定的功能。在具体实现的时候，这些功能一般是用函数的形式来实现的。那么这些函数之间的组织结构是怎么样的呢？如图 3-4 所示，主要有两种组织结构，即分层结构和混合结构。

所谓分层结构，就是把设备驱动程序当中的所有函数分为两种类型：一种是直接跟硬件打交道的，直接去操作和控制硬件设备，这些函数称为硬件接口；另一种是跟上层软件（包括操作系统、中间件和应用软件）打交道的，作为上层软件的调用接口。这些函数虽然也是设备驱动程序的一部分，但它们并不会直接去跟硬件打交道，它们的用途主要是作为上层软件的调用接口。在具体实现时，它们会去调用硬件接口当中的函数。

所谓混合结构，就是在设备驱动程序当中，上层接口和硬件接口的函数是混在一起、相互调用的，之间没有明确的层次关系。

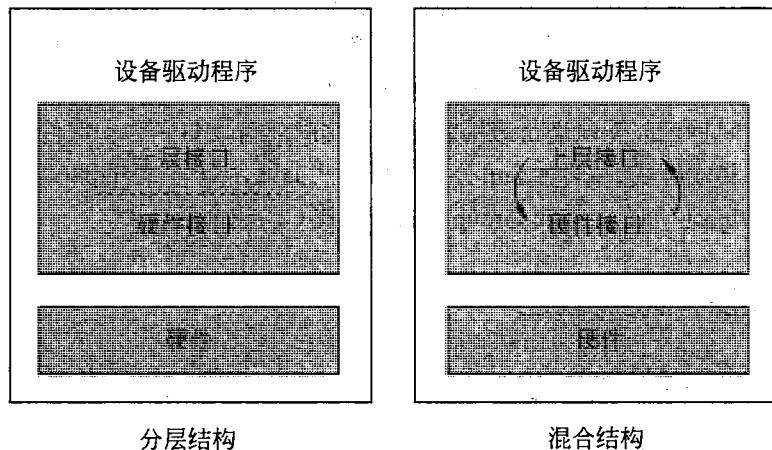


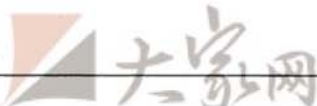
图 3-4 设备驱动程序的结构

分层结构的优点是：把所有与硬件有关的细节都封装在硬件接口当中，这样，如果将来硬件要升级，需要更新设备驱动程序，那么只需要改动硬件接口当中的函数即可，而上层接口当中的函数不用做任何修改。另外，无论是分层结构还是混合结构，它们给上层软件提供的调用接口都应该是明确而稳定的，即便设备驱动程序的内部有任何变化，也不会影响到上层软件，这样，在移植操作系统和应用程序的时候，就非常方便。

### 3.1.5 嵌入式中间件

中间件是 20 世纪 80 年代末提出的一种软件平台技术，经过十多年的发展，在银行、证券、电信等行业的大型计算机应用系统中得到了广泛的应用。近年来，在嵌入式系统中，





处理器的性能不断提高,系统的功能更为复杂,嵌入式软件对可靠性、实时性的要求越来越高,因此,中间件技术也被引入到嵌入式系统的设计当中,并与实时多任务操作系统紧密结合。这种全新的软件设计方法,可以使用户把精力集中到系统功能的实现上,从而真正实现嵌入式系统的软硬件协同设计。当然,在一个实际的嵌入式系统当中,对于是否要采用嵌入式中间件,这完全取决于具体的应用需求。

所谓嵌入式中间件,简单地说,就是在操作系统内核、设备驱动程序和应用软件之外的所有系统软件。嵌入式中间件的基本思路是:把原本属于应用软件层的一些通用的功能模块抽取出来,形成独立的一层软件,从而为运行在它上面的那些应用软件提供一个灵活、安全、移植性好、相互通信、协同工作的平台。这样,就可以有效地实现软件的可重用,降低应用软件的复杂性,提高系统的开发效率,缩短系统的开发周期,节约开发成本和维护费用,同时还保证了系统的高伸缩性、易升级性和稳定性。当然,如果在嵌入式系统中引入中间件,将会带来额外的开销,可能会对系统的性能造成一定的影响。

嵌入式中间件可以分为不同的类型,如消息中间件、对象中间件、远程过程调用(Remote Procedure Calls, RPC)、数据库访问中间件、安全中间件等。有些嵌入式系统较为复杂,单个中间件可能无法满足应用的需求,这时就需要将多种中间件集成在一起。市场上就有这样的集成解决方案,如 Sun 公司的嵌入式 Java、微软公司的 .NET Compact Framework、OMG (Object Management Group) 的嵌入式 CORBA 等。

## 3.2 嵌入式操作系统概述

### 3.2.1 嵌入式操作系统的概念

嵌入式操作系统(Embedded Operating System, EOS)就是工作在嵌入式环境中的操作系统。在 20 世纪 60 年代,嵌入式操作系统首先出现在国防系统中,并于 20 世纪七八十年代逐渐进入工业控制领域,经过几十年的发展,目前已广泛应用在工业、交通、能源、通信、医疗卫生、国防、日常生活等诸多领域。国际市场上已经出现了几十种嵌入式操作系统产品。

与通用的操作系统一样,我们也可以从两个方面来描述嵌入式操作系统的功能。

- 从软件开发的视角,可以把 EOS 看成是一种扩展机或虚拟机。它把底层的硬件细节封装起来,为运行在它上面的软件(如中间件软件和各种应用软件)提供了一个抽象的编程接口。软件的开发不是直接在机器硬件的层面上进行,而是在这个编程接口的层面上进行,这样就使得这些软件的开发变得更加容易。这里所说的编程接口,实际上就是操作系统对外提供的系统调用函数。因此,从这个角度来

看,就好像是操作系统对机器硬件进行了一个扩展,给用户提供了一层易于编程的虚拟的机器。

- 从系统管理的角度,可以把 EOS 看成是系统资源的管理者,负责管理系统当中的各种软硬件资源,如处理器、内存、各种 I/O 设备、文件和数据等,使得整个系统能够高效、可靠地运转。

作为运行在嵌入式环境中的操作系统, EOS 除了具有通用操作系统的基本功能之外,还有一些与嵌入式系统密切相关的特点:

- 其目标是为了完成某一项或有限项功能,而非通用型的操作系统。
- 在性能和实时性方面可能有严格的限制。
- 能源、成本和可靠性通常是影响设计的重要因素。
- 占用资源少,适合在有限存储空间运行。
- 系统功能可针对需求进行剪裁、调整,以便满足最终产品的设计要求。

对于不同的嵌入式操作系统,它们所包含的组件可能各不相同,但是一般来说,所有的操作系统都会有一个内核(kernel)。所谓的内核,是指系统其中的一个组件,它包含了 OS 的主要功能,即 OS 的各种特性及其相互之间的依赖关系。这些功能包括:任务管理、存储管理、输入/输出(I/O)设备管理和文件系统管理。

任务管理的主要功能是对嵌入式系统中的运行软件进行描述和管理,并完成处理机资源的分配与调度。存储管理的主要目标是如何来提高内存的利用率,方便用户的使用,并提供足够的存储空间。I/O 设备管理的主要目标是方便设备的使用,提高 CPU 和输入/输出设备的利用率;文件管理主要是解决文件资源的存储、共享、保密和保护等问题。对于不同的嵌入式操作系统,它们的内核设计也是各不相同的,并不一定要包含所有的四个功能模块,而取决于系统的设计以及实际的应用需求。

### 3.2.2 嵌入式操作系统的分类

嵌入式操作系统可以按照不同的标准来进行分类,例如,可以按照系统的类型、响应时间和软件结构来分类。

#### 1. 按系统的类型分类

按照系统的类型,可以把嵌入式操作系统分为三大类:商用系统、专用系统和开源系统。

- 商用系统:商业化的嵌入式操作系统。其特点是功能强大、性能稳定、应用范围相对较广,而且辅助软件工具齐全,可以胜任许多不同的应用领域。但商用系统的价格通常比较昂贵,如果用于一般的产品,会提高产品的成本从而失去竞争力。其典型代表是风河公司(WindRiver)的 VxWorks、微软公司的 Windows CE、Palm

公司的 PalmOS 等。

- 专用系统：一些专业厂家为本公司产品特制的嵌入式操作系统。这种系统一般不提供给应用开发者使用。
- 开源系统：开放源代码的嵌入式操作系统。这是近年来发展迅速的一类操作系统，其典型代表是  $\mu$ C/OS 和各类嵌入式 Linux 系统。开源系统具有免费、开源、性能优良、资源丰富、技术支持强等优点，在信息家电、移动通信、网络设备和工业控制等领域得到了越来越广泛的应用。

## 2. 按响应时间分类

按照系统对响应时间的敏感程度，可以把嵌入式操作系统分为两大类：实时操作系统（Real Time Operating System, RTOS）和非实时操作系统。

顾名思义，实时操作系统就是对响应时间要求非常严格的系统，当某一个外部事件或请求发生时，相应的任务必须在规定的时间内完成相应的处理。实时系统的正确性不仅依赖于系统计算的逻辑结果，还依赖于产生这些结果所需要的时间。

实时操作系统可以分为硬实时和软实时两种情形。

- 硬实时系统：系统对响应时间有严格的要求，如果响应时间不能满足，这是绝不允许的，可能会引起系统的崩溃或致命的错误。
- 软实时系统：系统对响应时间有要求，如果响应时间不能满足，将带来额外的代价，不过这种代价通常能够接受。

非实时系统在响应时间上没有严格的要求，如分时操作系统，它是基于公平性原则的，各个进程分享处理器，获得大致相同的运行时间。当一个进程在进行 I/O 操作时，会交出处理器，让其他的进程运行。

## 3. 按软件结构分类

按照软件的体系结构，可以把嵌入式操作系统分为三大类：单体结构、分层结构和微内核结构。它们之间的差别主要表现在两个方面：一是内核的设计，即在内核中包含了哪些功能组件；二是在系统中集成了哪些其他的系统软件（如设备驱动程序和中间件）。

### （1）单体结构

单体结构（monolithic）是一种常见的组织结构。在单体结构的操作系统中，中间件和设备驱动程序通常就集成在系统内核当中。整个系统通常只有一个可执行文件，里面包含了所有的功能组件，如图 3-5 所示。系统的结构就是无结构，整个操作系统由一组功能模块组成，这些功能模块之间可以相互调用。例如，嵌入式 Linux 操作系统、Jbed RTOS、 $\mu$ C/OS-II 和 PDOS 都属于单体内核系统。

单体结构的优点是性能较好，系统的各个模块之间可以相互调用，通信开销比较小。它的缺点是：操作系统具有体积庞大、高度集成和相互关联等特点，因而在系统剪裁、修

改和调试等方面都较为困难。

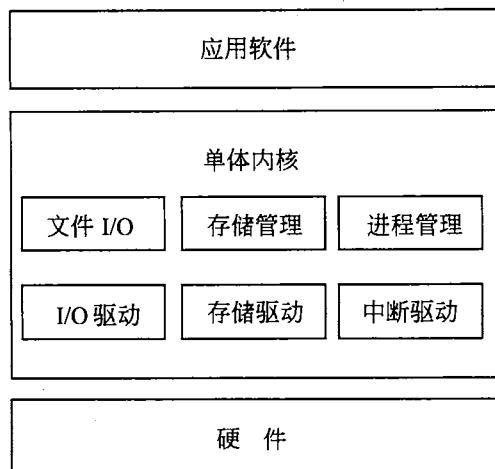


图 3-5 单体结构

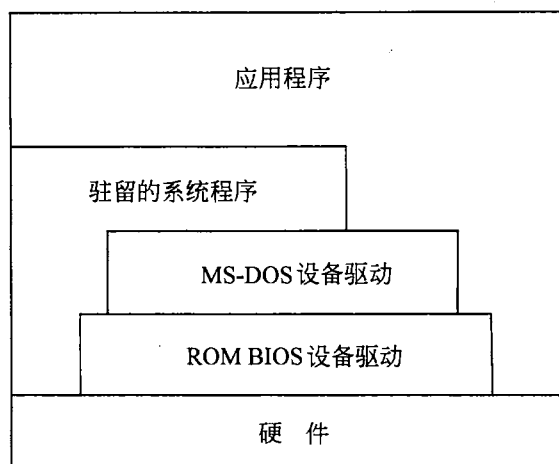


图 3-6 分层结构

## （2）分层结构

在分层结构（layered）中，一个操作系统被划分为若干个层次（0~N），各个层次之间的调用关系是单向的，即某一层次上的代码只能调用比它低层的代码。与单体结构相似，分层结构的操作系统也是只有一个大的可执行文件，其中包含有设备驱动程序和中间件。由于采用了层次结构，所以系统的开发和维护都较为简单。当我们要替换系统当中的某一层时，不会影响到其他的层次。但是，这种结构要求在每个层次上都要提供一组 API 接口函数，这就会带来额外的开销，从而影响到系统的规模和性能。图 3-6 所示的是 MS-DOS 的结构，这是一个有代表性的、组织良好的分层结构。

## （3）微内核结构

微内核（microkernel）结构，或者说客户/服务器结构（client/server）的操作系统，指的是在内核当中，把操作系统的大部分功能都剥离出去，只保留最核心的功能单元（如进程管理和存储管理）。微内核结构的特点就是内核非常小，大部分的系统功能都位于内核之外，例如，所有的设备驱动程序都被置于内核之外，如图 3-7 所示。

在微内核操作系统中，新的功能组件可以被动态地添加进来，所以它具有易于扩展、调试方便等特点。另外，由于大部分的系统功能被放置在内核之外，而且客户单元和服务单元单元的内存地址空间是相互独立的，所以系统的安全性更高。它还有一个优点就是移植方便。但是，与其他类型的操作系统相比（如单体内核），微内核操作系统的运行速度可能会慢一些，这是因为核内组件与核外组件之间的通信方式是消息传递，而不是直接的函数

调用。另外，由于它们的内存地址空间是相互独立的，所以在切换的时候，也会增加额外的开销。许多嵌入式操作系统采用的都是微内核的方式，如 OS-9、C Executive、VxWorks、CMX-RTX、Nucleus Plus 和 QNX 等。

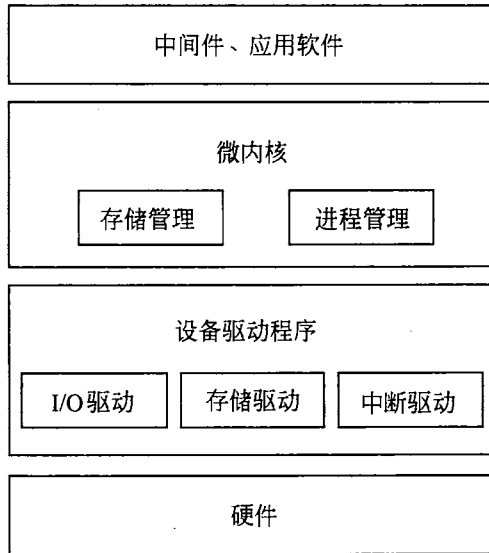


图 3-7 微内核结构

### 3.2.3 常见的嵌入式操作系统

随着嵌入式系统在国民经济各个领域的广泛应用，嵌入式操作系统也得到了蓬勃的发展。从早期的实模式进化到保护模式，从微内核技术进化到超微内核技术，从支持单处理器发展到支持多处理器、分布式和实时网络，嵌入式操作系统已经成为操作系统研究领域中的一个重要分支。目前，国内外已经有数十家公司在从事相关方面的研究，开发了数以百计的各具特色的嵌入式操作系统产品，其中比较有影响系统包括 VxWorks、嵌入式 Linux、Windows CE、 $\mu$ C/OS-II 和 PalmOS 等。

#### (1) VxWorks

VxWorks 是美国 WindRiver System 公司开发的一款嵌入式实时操作系统，具有良好的可靠性和卓越的实时性，是目前嵌入式系统领域中使用最广泛、市场占有率最高的商业系统。VxWorks 支持各种主流的 32 位处理器，如 x86、Motorola MC68xxx、Coldfire、PowerPC、MIPS、ARM、i960 等。它基于微内核的体系结构，整个系统由四百多个相对独立、短小精练的目标模块组成，用户可以进行裁减和配置，根据自己的需要来选择适当的模块。

VxWorks 采用 GNU 类型的编译和调试器，它的大多数 API 函数都是专有的。

VxWorks 操作系统主要由以下几个功能模块组成。

- 高效的实时微内核 Wind：这是 VxWorks 的核心，它包括基于优先级的任务调度、任务间的通信、同步和互斥、中断处理、定时器和内存管理机制等。
- I/O 处理系统：VxWorks 提供了一个快速灵活的与 ANSI C 兼容的 I/O 系统，包括 UNIX 标准的缓冲 I/O 和 POSIX 标准的异步 I/O。
- 文件系统：VxWorks 提供了适合于实时应用的文件系统，主要包括与 MS-DOS 兼容的文件系统、与 RT-11 兼容的文件系统、Raw Disk 文件系统和 SCSI 磁带设备。
- 网络处理模块：能与许多运行其他协议的网络进行通信，如 TCP/IP、NFS、UDP、SNMP、FTP 等。
- 虚拟内存模块 VxVMI：主要用于对指定内存区的保护，以加强系统的安全性。
- 板级支持包 BSP：是系统用来管理硬件的功能模块，对各种板卡的硬件功能提供了统一的接口，它由初始化和驱动程序两部分组成。

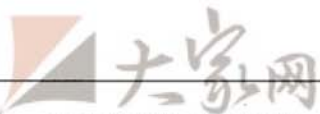
## (2) 嵌入式 Linux

Linux 是 1991 年由芬兰人 Linus Torvalds 发明的，从诞生到现在短短十几年的时间，Linux 已经发展成为一个功能强大、设计完善的操作系统，不仅在通用操作系统领域与 Windows 等商业系统分庭抗争，而且在新兴的嵌入式操作系统领域也获得了飞速的发展。嵌入式 Linux (Embedded Linux) 是指对标准 Linux 进行小型化剪裁处理之后，可固化在存储器或单片机中，适合于特定嵌入式应用场合的专用 Linux 操作系统。

嵌入式 Linux 的开发和研究已经成为操作系统领域的一个热点，其特点包括以下几条。

- 高性能、可裁剪的内核：Linux 内核的高效和稳定已经在各个领域得到了验证，其独特的模块机制使用户可以根据自己的需要，实时地将某些模块插入到内核或从内核中移走，很适合于嵌入式系统的小型化的需要。
- 完善的网络通信和文件管理机制：Linux 支持所有标准的 Internet 网络协议，并且很容易移植到嵌入式系统当中。此外，Linux 还支持 ext2、fat16、fat32、romfs 等文件系统，这些都为嵌入式应用的开发打下了很好的基础。
- 优秀的开发工具：一套完善的开发和调试工具是嵌入式系统开发的关键。嵌入式 Linux 提供了一套完整的工具链 (Tool Chain)，它利用 GNU 的 gcc 做编译器，用 gdb、kgdb、xgdb 做调试工具，能够方便地实现从操作系统到应用软件各个级别的调试。
- 免费、开放源码：Linux 是开放源码的自由操作系统，用户可以根据自己的应用需要方便地对内核进行修改和优化，这对于千差万别的嵌入式系统来说是非常重要的。





- 广泛的硬件支持：支持 x86、ARM、MIPS、Alpha、PowerPC 等多种体系结构，目前已经成功移植到数十种硬件平台，几乎能够运行在所有流行的 CPU 上，支持各种主流硬件设备和最新硬件技术。
- 软件资源丰富：几乎每一种通用程序在 Linux 上都能找到，从而减轻了开发工作量。

常见的嵌入式 Linux 包括  $\mu$ Clinux、RT-Linux、Embedix 和 Hard Hat Linux 等。 $\mu$ Clinux 主要针对没有 MMU 的微处理器；RT-Linux 是最早实现硬实时支持的 Linux 版本；Embedix 的设计使用了模块化的设计方案，方便系统剪裁；Hard Hat Linux 是 MontaVista 公司开发的一个嵌入式实时系统，可以针对硬件环境进行配置，以获得最佳的性能和最小的容量。

### (3) Windows CE

Windows CE 是微软公司发布的嵌入式操作系统，主要用在个人数字助理（Personal Digital Assistant, PDA）和智能电话（SmartPhone）等个人手持终端上。Windows CE 是一个基于优先级的多任务操作系统，提供了 256 个优先级别，但它并不是一个硬实时系统。Windows CE 操作系统的基本内核需要至少 200KB 的 ROM，它支持 Win 32 API 子集、支持多种用户界面硬件、支持多种串行和网络通信技术。Windows CE 主要包含五个功能模块。

- 内核模块：支持进程和线程处理及内存管理等基本服务。
- 内核系统调用接口模块：允许应用软件访问操作系统提供的服务。
- 文件系统模块：支持 DOS 等格式的文件系统。
- 图形窗口和事件子系统模块：控制图形显示，并提供 Windows GUI 图形界面。
- 通信模块：允许同其他的设备进行信息交换。

Windows CE 操作系统最大的特点是能提供与 PC 机类似的图形界面和主要的应用程序——集成了大量的 Windows XP Professional 的特性，包括桌面、任务栏、窗口、图标、控件和各种应用程序。这样，只要是对 PC 机上的 Windows 操作系统比较熟悉的用户，可以很快地使用基于 Windows CE 的嵌入式设备。另外，微软公司提供了一组功能强大的应用程序开发工具，如 Visual Studio.NET、Embedded Visual C++、Embedded Visual Basic 等，专门用于对 Windows CE 操作系统的开发，程序员可以方便地使用这些工具开发出丰富多彩的嵌入式应用软件。

### (4) $\mu$ C/OS-II

$\mu$ C/OS 是美国人 Jean Labrosse 在 1992 年开发的一个嵌入式操作系统，并于 1998 年推出了它的升级版  $\mu$ C/OS-II。 $\mu$ C/OS-II 是一种免费、开放源代码、结构小巧、基于可抢占优先级调度的实时操作系统，其内核提供任务调度与管理、时间管理、任务间同步与通信、内存管理和中断服务等功能。

$\mu$ C/OS-II 主要面向中小型嵌入式系统，具有执行效率高、占用空间小、实时性能优良

和可扩展性强等特点, 最小内核可编译至 2KB, 一般情形下占用内存在 10KB 数量级。它的内核本身并不支持文件系统, 但它具有良好的扩展性能, 如果需要的话可以自行加入。由于免费、源码开放、规模较小,  $\mu\text{C}/\text{OS-II}$  不仅在众多的商业领域中获得了广泛的应用, 而且被许多大学所采纳, 作为教学用的嵌入式实时操作系统。

### (5) PalmOS

PalmOS 是 Palm 公司开发的一种 32 位的嵌入式操作系统, 主要应用在 PDA 和手机等手持移动终端上, 是市场占有率最高的 PDA 操作系统。PalmOS 的优点是功能强大、性能稳定、设计简捷、效率高, 而且第三方应用程序非常丰富, 到目前为止, 已有多达两万多个应用软件运行在 PalmOS 操作系统上。

## 3.3 任务管理

### 3.3.1 多道程序技术

嵌入式操作系统可以分为两种类型: 单道程序设计和多道程序设计 (multi-programming)。所谓单道程序设计, 就是在操作系统当中, 在任何时候只能有一个程序在运行。所谓多道程序设计, 就是在操作系统当中, 允许多个程序同时存在并运行。在现代操作系统当中, 为了提高系统资源的利用率, 普遍采用了多道程序技术。

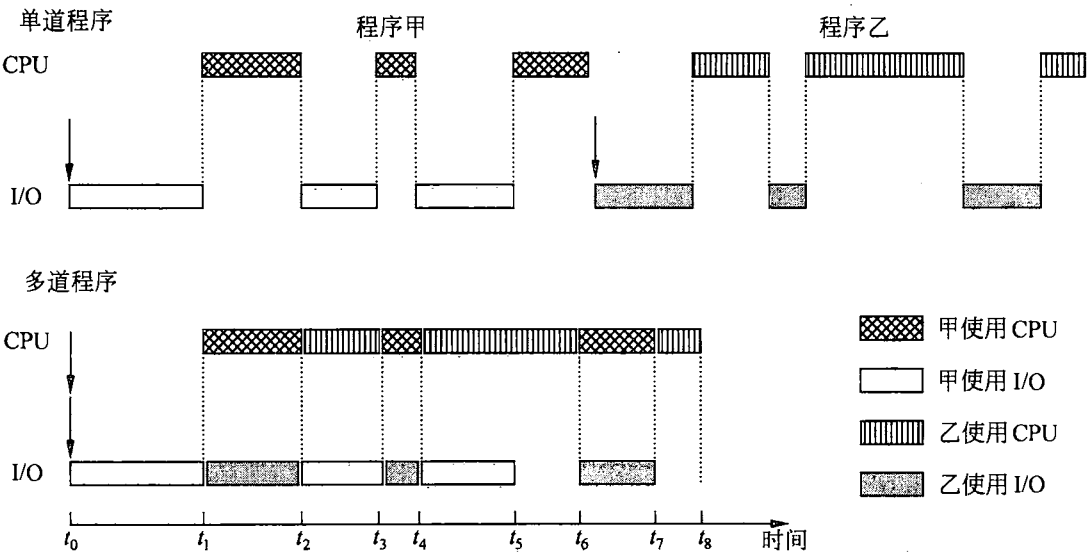
图 3-8 所示是单道和多道程序设计的一个例子。有两个程序甲和乙, 它们在运行过程中都要用到 CPU 和 I/O 设备。如图 3.8 所示, 我们用不同的方框来表示这两个程序对两种资源的使用情况, 方框的长度表示使用的时间。

在单道程序设计的环境下, 在任何时候, 系统中只能有一个程序在运行, 因此, 这两个程序的执行只能是一个接一个。首先执行程序甲, 从  $t_0$  时刻开始, 到  $t_6$  时刻结束。然后再执行程序乙, 从  $t_6$  时刻开始, 一直到它所有的工作都已完成。

在多道程序设计的环境下, 允许多个程序同时运行, 当一个程序在访问 I/O 设备时, 会主动把 CPU 交出来, 让另一个程序去运行, 这样就提高了系统资源的使用效率。具体来说, 首先, 在  $t_0$  时刻, 甲和乙都打算运行, 它们的第一件事情都是进行 I/O 操作, 由于资源有限, 只能满足一个程序的请求。假设甲的请求得到满足, 它先开始执行, 从  $t_0$  到  $t_1$ , 甲一直在使用 I/O 设备, 在此期间, 乙一直处于等待状态。在到达了  $t_1$  时刻后, 甲已经执行完了 I/O 操作, 下一步要执行一小段 CPU 操作, 这样它就把刚刚占用的 I/O 设备释放出来, 交给程序乙去使用。因此, 在  $t_1$  到  $t_2$  期间, 程序甲在使用 CPU, 程序乙在使用 I/O 设备, 它们互不影响。在到达  $t_2$  时刻后, 甲又要执行一小段 I/O 操作, 而乙恰巧要执行一小段 CPU 操作, 因此, 在  $t_2$  到  $t_3$  期间, 它们相互交换了资源, 继续执行。同样的情形也发



生在  $t_3$  时刻和  $t_4$  时刻。但是在  $t_5$  时刻，甲已经使用完了 I/O 设备，而乙仍然在使用 CPU，所以甲只能处于等待状态，等到  $t_6$  时刻再交换资源。这样一直进行下去，在  $t_7$  时刻，甲执行完毕，在  $t_8$  时刻，乙也执行完毕。显然，在多道程序设计的操作系统中，由于 CPU 和 I/O 设备的使用是并行进行的，所以在总的执行时间上要明显少于单道程序系统。



3.3.2 进程、线程和任务

1. 进程

在多道系统当中，允许多个程序同时存在，各个程序之间是并发执行的，它们共享系统的资源，CPU 需要在各个运行的程序之间来回地切换，不断地从一个程序切换到另一个程序。这样一来，仅仅依靠静态的“程序”这个概念，要想正确地描述这些多道的并发活动过程就变得非常的困难。为此，必须提出一种新的概念实体，即进程（process）。

进程的概念是 20 世纪 60 年代由美国麻省理工学院的研究人员在开发著名的 Multics 操作系统时提出来的。所谓的进程，简单地说，一个进程就是一个正在运行的程序。这个定义虽然很简单，但却有丰富的内涵。一般来说，一个进程至少应该包括以下几个方面的内容：

- 相应的程序：进程既然是一个正在运行的程序，当然需要有相应程序的代码和数据。
- CPU 上下文：程序在运行时，CPU 中含有各种寄存器的当前值，包括程序计数器

(Program Counter, PC), 用于记录将要取出的指令的地址; 程序状态字 (Program Status Word, PSW), 用于记录处理器的运行状态信息; 通用寄存器, 用于存放数据或地址; 段寄存器, 用于存放程序中各个段的地址; 栈指针寄存器, 用于记录栈顶的当前位置。

- 一组系统资源: 包括操作系统用来管理进程的数据结构、进程的内存地址空间、进程正在使用的文件等。

总而言之, 进程包含了正在运行的一个程序的所有状态信息。

进程和程序是两个既有联系又有区别的概念, 不能把两者混为一谈。它们之间的关系主要表现在以下方面。

- 一个程序主要由两部分内容组成——代码和数据, 它是一个静态的概念。而进程是正在执行的程序, 它也由两部分内容组成: 程序和该程序的运行上下文, 它是一个动态的概念。如果把一部动画片的电影副本比拟成一个程序, 那么这部动画片的一次放映过程就是一个进程。
- 进程和程序之间并不是一一对应的。一个进程在运行的时候可以启动一个或多个程序, 例如, 当一个进程在进行 C 源程序编译时, 需要执行前处理、词法语法分析、代码生成和优化等几个功能步骤, 每个步骤可以用相互独立的程序来完成, 这样, 在一个进程中就启动了多个不同的程序。反之, 同一个程序也可能由多个进程同时执行, 例如, 我们可以同时启动多个 Word 字处理器来编辑不同的文档, 对于 Word 程序而言, 它只有一份, 但它每运行一次就创建了一个新的进程。
- 程序可以作为一种软件资源长期保存, 以文件的形式存放在硬盘或光盘上, 而进程则是一次执行过程, 它是暂时的, 是动态地产生和终止的。仍以电影播放为例, 电影副本是可以长期保存的, 而一次放映活动却只延续一两个小时。

图 3-9 通过一个例子阐述了进程与程序之间的区别与联系。图的左边是一个 C 语言程序, 该程序有两个函数: 主函数 main 和子函数 A。当这个程序开始运行后, 就创建了一个相应的进程 (即图的右边)。在这个进程当中, 除了静态的程序以外, 还增加了一些新的内容, 包括堆、栈、CPU 寄存器等, 这些都是程序在运行时所产生的运行上下文。

进程有三个特性。

- 动态性: 进程是一个正在运行的程序, 而程序的运行状态是在不断地变化的。例如, 当一个程序在运行的时候, 每执行完一条指令, PC 寄存器的值就会增加, 指向下一条即将执行的指令。而 CPU 中用来存放数据和地址的那些通用寄存器, 它们的值肯定也不断变化。另外, 堆和栈的内容也在不断变化, 每当发生一次函数调用时, 就会在栈中分配一块空间, 用来存放此次函数调用的参数和局部变量,

而当函数调用结束后，这块栈空间就会被释放掉。总之，一切都在变化当中，不变的只有一些只读的内容，如程序的代码。

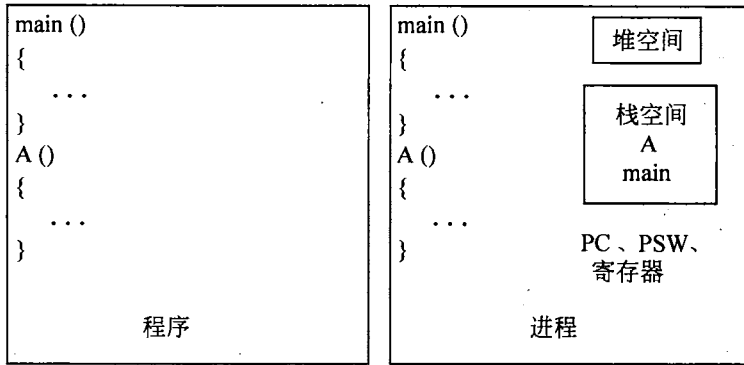


图 3-9 程序与进程

- 独立性：一个进程是一个独立的实体，是计算机系统资源的使用单位。每个进程都有自己的运行上下文和内部状态，在它运行的时候独立于其他的进程。
- 并发性：从宏观上来看，在系统中同时有多个进程存在，它们相互独立地运行。

图 3-10 所示表示了四个进程 A、B、C、D 在系统中并发地运行。从图 3-10 中可以看出，虽然从宏观上来说，这四个进程都在系统中运行，但实际上，从微观上来看，在任何一个特定的时刻，只有一个进程在 CPU 上运行，也就是说，各个进程实际上是一个接一个地顺序运行。因为 CPU 只有一个，所以在任何时刻最多只能有一个进程去使用它。从时间上来看，开始是进程 A 在运行，然后是进程 B 在运行，之后是进程 C 和进程 D。接下来又轮到了进程 A 去运行，之后是进程 B、C 和 D。因此，在单 CPU 的情形下，所谓的并发性，指的是宏观上并发运行，而微观上还是顺序运行，各个进程轮流去使用 CPU 资源。在具体实现上，以 CPU 当中的程序计数器 PC 为例，真正的、物理上的 PC 寄存器只有一个，当四个进程在轮流执行时，PC 的取值也在不断地发生变化，其运动轨迹是先在进程 A 内部流动，然后再到进程 B 的内部流动，再到进程 C 和 D。但是从进程的独立性的角度来说，每个进程都有“自己”独立的 PC 寄存器，即逻辑上的 PC 寄存器，它们的取值的运动轨迹是在各个进程的内部，从上往下，而且相互独立、互不影响。所谓的逻辑 PC，其实就是一个内存变量。当一个进程要运行的时候，就把它的逻辑 PC 中的值装入到物理 PC 当中去；反之，当一个进程暂且不运行的时候，就把物理 PC 中的值保存在它的逻辑 PC 当中。例如，在本图中，当进程 A 要执行的时候，就把 A 的逻辑 PC 的值复制到物理 PC 中，然后开始运行。后来当轮到 B 运行的时候，先把物理 PC 的当前值保存到 A 的逻辑 PC 中，然后再

把 B 的逻辑 PC 的值装入到物理 PC 中, 即可运行。这样就实现了各个进程的轮流运行。另外, 这个例子只考察了 PC 寄存器这种资源, 实际上在一个进程的运行上下文当中有很多其他的资源, 如 PSW 寄存器、各种通用寄存器、栈指针寄存器等, 它们的使用方式都是类似的, 即物理上的硬件寄存器只有一个, 但每一个进程都有一个相互独立的逻辑上的寄存器。

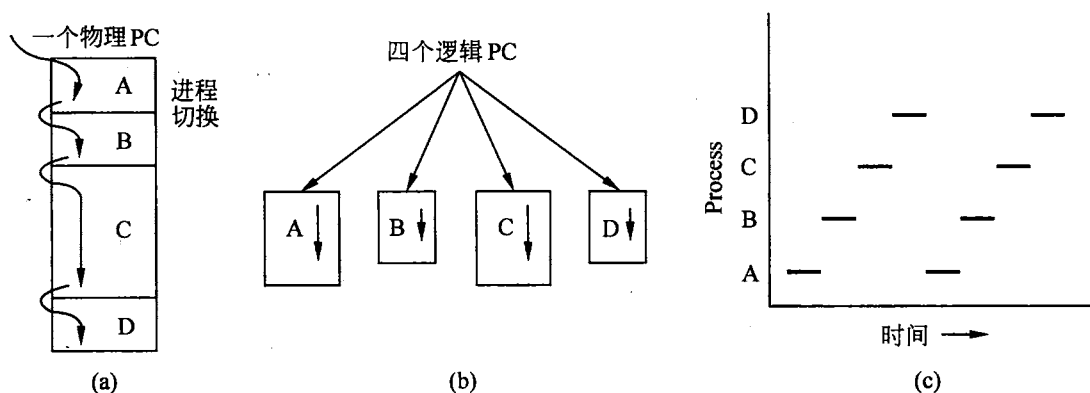


图 3-10 四个进程在并发运行

## 2. 线程

自从 20 世纪 60 年代人们提出进程这个概念以来, 在操作系统当中一直都是以进程来作为独立运行的基本单位。直到 20 世纪 80 年代中期, 人们又提出了更小的能独立运行的基本单位, 也就是线程 (thread)。

线程的定义很简单, 所谓的线程, 就是进程当中的一条执行流程。为了进一步阐明线程的内涵, 有必要再对进程这个概念做一个回顾。如图 3-11 所示, 我们可以从两个不同的方面来理解进程: 一方面, 从资源组合的角度, 进程把一组相关的资源组合起来, 构成了一个资源平台, 或者说资源环境, 其中包括运行上下文、内存地址空间、打开的文件等, 在图中用深色的方框来表示; 另一方面, 从程序运行的角度, 进程就是一个正在运行的程序, 这是它的一个本质特征。从这个角度, 可以把进程看成是代码在这个资源平台上的一条执行流程, 也就是线程, 在图中用一条带有箭头的线段来表示。当进程这个概念刚刚被提出来的时候, 这两者的关系是密不可分、一一对应的。每一个进程都会提供这样的一个资源平台, 同时, 在这个资源平台上, 有且仅有一条执行流程。因此, 人们一般也不会去对这两者加以区分, 当谈到进程的时候, 既是指它的资源平台, 又是指它的执行流程。后来, 由于实际的需要, 人们觉得有必要把这两者分隔开来, 资源平台就是资源平台, 而

代码的执行流程就称为线程。

按照这个思路，我们可以重新定义进程：进程等于线程加上资源平台。这样做的好处是：

- 在一个进程当中，或者说在一个资源平台上，可以同时存在多个线程。如图 3-12 所示，在这个例子当中，一个进程包含有三个线程。
- 可以用线程来作为 CPU 的基本调度单位，使得各个线程之间可以并发执行。
- 对于同一个进程当中的各个线程来说，由于它们是运行在相同的资源平台上，所以它们可以共享该进程的各种资源，如内存地址空间、代码、数据、文件等，这就使得线程之间的通信与交流变得非常方便。

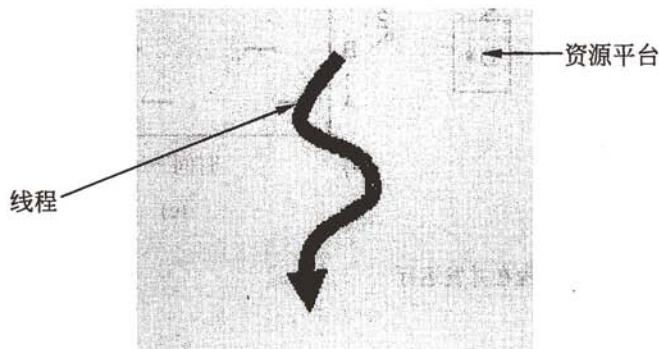


图 3-11 资源平台与线程

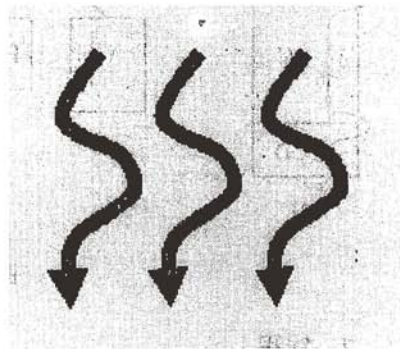


图 3-12 多线程

既然线程是代码在进程的资源平台上的一条执行流程，那么是不是进程的所有资源都能够共享呢？答案是否定的。对于同一个进程的各个线程，它们可以共享该进程的大部分资源，但也有一小部分资源是不能共享的，每个线程都必须拥有各自独立的一份。这些资源包括：CPU 运行上下文（如 PC 寄存器、PSW 寄存器、通用寄存器和栈指针等）和栈。

图 3-13 所示表示了线程与进程之间的资源关系。图 3-13(a)表示一个单线程的进程，上面是它所拥有的资源，包括代码、数据、文件、寄存器和栈等，下面是它的线程；图 3-13(b)表示一个多线程的进程，它包含有三个线程。从图中我们可以看出，在各种资源当中，代码、数据和文件等资源是进程一级的资源，被所有的线程所共享，因此只有一份。所有的线程都可以使用这些资源，并且通过这些资源来进行通信与交流，只有寄存器和栈这两种资源是线程所独有的，每个线程都有自己独立的一份，三个线程就有三份。当然，这里所说的三份资源指的是逻辑上的资源，而非物理上的硬件资源。例如，物理的 PC 寄存器只

有一个,但每个线程都可以拥有自己的一个逻辑PC。那么为什么这两种资源必须是独有的呢?为什么寄存器和栈不能共享?原因在于,对于一个线程而言,当它在运行的时候,这两种资源是必不可少的硬件资源,是不能与其他线程共享的。事实上,由于各个线程之间是并行执行的,而线程在执行的时候,必定会修改CPU中各个寄存器的内容,如PC寄存器、PSW寄存器、通用寄存器等,因此,这些内容都是线程的私有信息,各个线程之间是不能共享的。另外,当一个线程在运行的时候,也可能会发生函数的调用,这时就需要分配一块栈空间,用来存放此次函数调用的参数、上下文和局部变量,因此,独立的栈空间也是必不可少的。总而言之,在一个线程的运行过程中,最直接、最必不可少的资源就是CPU上下文和栈,这两种资源是不能与其他线程共享的,每个线程都必须有自己独立的一份。

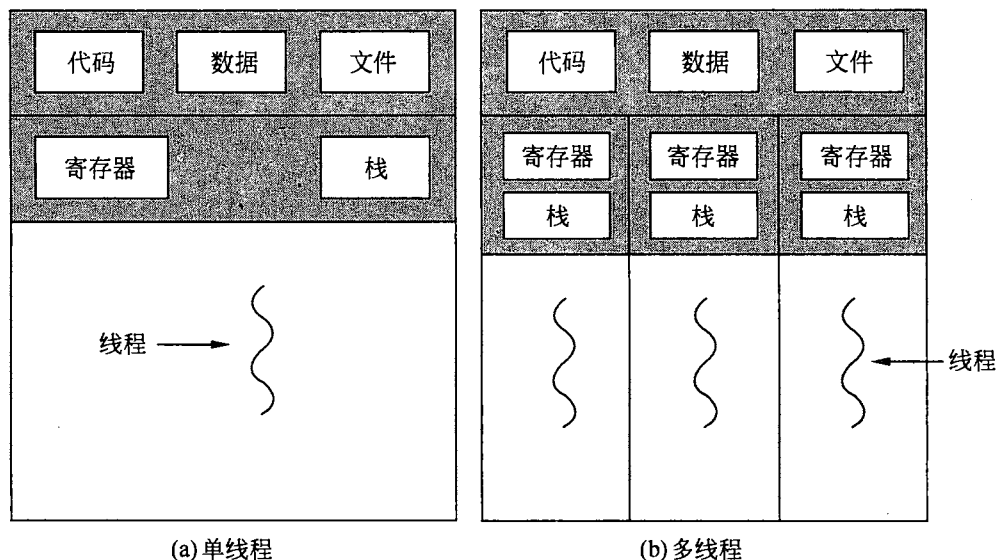


图 3-13 线程与进程的资源关系

### 3. 任务

与传统的操作系统不同,在许多嵌入式操作系统当中,并没有使用“进程”或“线程”这两个术语,而是把能够独立运行的实体称为“任务”(task)。那么这里所说的任务到底是进程还是线程呢?对于不同的系统,这个问题有不同的回答。因此,当我们在研究一个具体的嵌入式操作系统的时候,要注意加以区分。

下面来分析一个具体的例子。在一个实际的工程项目中,软件平台采用的是某一种实

时的嵌入式操作系统。该项目有两个.c 源文件,如图 3-14 和图 3-15 所示。src1.c 的功能是:任务 A 循环地从一个 SOCKET 当中接收数据;任务 B 每隔 100ms 向这个 SOCKET 发送一条响应消息,而这个定时的功能是由 src2.c 当中的任务 C 来实现的。任务 C 和任务 B 之间通过信号量来进行任务间的同步。现在需要分析该操作系统当中的“任务”的概念,到底是进程还是线程?

图 3-14 所示是源文件 src1.c。在这个文件当中,首先定义了两个全局变量: g\_nSocketId 表示 socket 标识; g\_synSemId 表示信号量标识,用来实现任务间的同步。接下来定义了一个初始化函数 testInit, 它首先创建 Socket, 建立连接, 即对 g\_nSocketId 这个全局变量进行赋值, 然后调用 taskSpawn 函数创建了 A 和 B 两个任务。taskSpawn 函数的功能就是用来

```
int    g_nSockId;        /* socket 标识, 全局变量 */
semId g_synSemId;        /* 信号量标识, 全局变量 */
void testInit(void) /* 初始化函数 */
{
    创建 SOCKET, 建立连接; /* g_nSockId 被赋值 */
    //taskSpawn: 创建一个任务。参数: 任务名, 优先级, 栈大小, 函数名, 函数的输入参数
    taskSpawn("tTestTskA", 50, 2000, testTskA, 0, ...); /*创建任务 A*/
    taskSpawn("tTestTskB", 50, 2000, testTskB, 0, ...); /*创建任务 B*/
}
void testTskA(void)
{
    char *pChRxBuf;
    pChRxBuf = malloc(100);
    while(1){
        recv(g_nSockId, pChRxBuf, ....);
        ....
    }
}
void testTskB(void)
{
    char pChTxBuf[100] = "Send message back every 100ms";
    while(1){
        semTake(g_synSemId);
        send(g_nSockId, pChTxBuf, ....);
    }
}
```

图 3-14 src1.c

创建一个任务，其参数包括：任务的名称、任务的优先级、任务所需要的栈空间的大小、任务的入口函数名、以及这个函数的输入参数等。接下来定义了一个函数 `testTaskA`，该函数就是任务 A 的入口函数。它首先定义了一个字符指针，并用它来动态申请了 100 个字节的内存空间，然后循环地从 SOCKET 当中接收数据，并保存在这个缓冲区当中。最后还定义了一个函数 `testTaskB`，该函数是任务 B 的入口函数，其功能就是每隔 100ms 向 SOCKET 发送一条响应消息。

图 3-15 所示是源文件 `src2.c`，它首先用一个 `extern` 语句，表明在这个源文件当中，需要用到源文件 `src1.c` 当中的全局变量 `g_synSemId`，即用于任务间同步的信号量标识。然后在 `test` 函数当中，首先创建了一个同步信号量，并且把它初始化为空。接下来创建了任务 C，函数 `testTaskC` 就是任务 C 的入口函数，其功能很简单：循环地去延迟 100 ms 的时间，等到 100 ms 过了以后，就通过信号量去唤醒任务 B。

```
extern semId g_synSemId;
void test(void)
{
    创建同步信号量，并初始为空； /* 即使用全局变量 g_synSemId */
    taskSpawn("tTestTskC", 50, 2000, testTskC, 0, ....); /*创建任务 C*/
}
void testTskC(void)
{
    while(1){
        taskDelay(100); /* 延时 100ms，同时交出 CPU 资源 */
        semGive(g_synSemId);
    }
}
```

图 3-15 src2.c

通过对这两段程序的分析，我们认为，在这个嵌入式操作系统当中，它的“任务”其实就是线程。原因主要有两点：

- 从任务的创建过程来看，它所需要的参数主要是任务的优先级、栈空间的大小和函数名。换言之，任务具有独立的优先级和栈空间，这些都是创建一个线程所必需的资源，而 CPU 上下文一般也是存放在栈空间当中的。
- 在本例当中，对于不同的任务，它们能够访问相同的全局变量，例如，任务 A 和任务 B 都使用了 `g_nSockeId` 这个全局变量；任务 B 和任务 C 都使用了同步信号量这个全局变量。这说明，在这些任务之间，可以很方便地、直接地去使用共享的内存单



元，而不需要经过系统内核来进行通信。而这正好就是线程的特点：同一个进程中的所有线程可以共享该进程当中的各种资源，包括内存地址空间和文件等资源。

以上这个例子当中的嵌入式操作系统实际上是 VxWorks，它的“任务”其实就是线程。类似的系统还有  $\mu\text{C}/\text{OS-II}$ 、Jbed 等。当然，也有一些嵌入式操作系统，如一些嵌入式 Linux 系统，它里面的任务指的是进程。为了方便起见，本书将按照惯例统一使用“任务”这个名词术语，并在需要的时候指明其是进程还是线程。

### 3.3.3 任务的实现

#### 1. 任务的层次结构

在多道程序的嵌入式操作系统中，同时存在着多个任务，这些任务之间的结构一般为层状结构，存在着父子关系。当嵌入式内核刚刚启动的时候，只有一个任务存在，然后由该任务派生出所有其他任务，如图 3-16 所示。

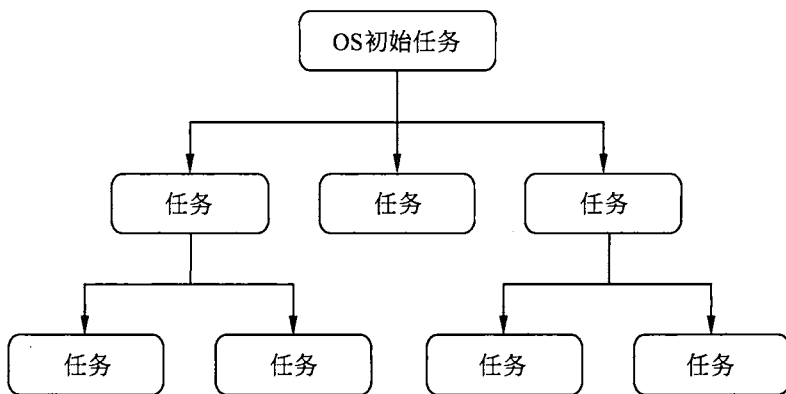


图 3-16 任务的层次结构

#### 2. 任务的创建与终止

在一个嵌入式操作系统当中，任务的创建主要发生在以下三种情形下。

- 系统初始化：当嵌入式内核在进行系统初始化的时候，一般都会创建一些任务。例如，它可能会创建一些前台任务，负责与嵌入式系统的用户进行交互，也可能会创建一些后台任务，这些任务不直接跟用户打交道，而是在后台完成一些特定的功能，如键盘扫描、系统状态检测、时间统计等。
- 任务运行过程中：除了在系统初始化的时候会创建任务以外，当一个任务正在运行的时候，如果需要的话，也能够使用相应的系统调用来创建新的任务，以帮助它完成自己的工作。

- 用户提出请求：在一些具有交互功能的嵌入式系统中，用户可以通过输入命令或单击图标的方式，让系统启动一个新的任务。例如，在一个 PDA 中，用户可以单击某一个游戏，或打开视频播放器，这时系统就会创建相应的任务来满足用户的请求。

虽然在以上这三种情况下，都能够创建一个新的任务，但是从技术的角度来说，实际上只有一种创建任务的方法，也就是在一个已经存在的任务当中，通过调用相应的系统函数来创建一个新的任务。

在嵌入式操作系统当中，任务的创建主要有两种可能的实现模型，即 `fork/exec` 和 `spawn`。两者既有联系又有区别。

- `fork/exec` 模型源于 IEEE/ISO POSIX 1003.1 标准，而 `spawn` 模型是从它派生出来的。这两种模型在创建任务的时候，过程非常相似。首先为新任务分配相应的数据结构，存放它的各种管理信息，然后为它分配内存空间，存放任务的代码和数据。当这个新任务准备就绪后，就可以启动它运行了。
- 两种模型的差别主要在于内存的分配方式。在 `fork/exec` 模型下，首先调用 `fork` 函数为新任务创建一份与父任务完全相同的内存空间，然后再调用 `exec` 函数装入新任务的代码，并用它来覆盖原有的属于父任务的内容。这样做的好处是：对于新创建的子任务来说，如果需要的话，它可以从父任务那里继承代码、数据等各种属性。而在 `spawn` 模型下，摒弃了继承这一功能，在创建新任务的时候，直接为它分配一个全新的地址空间，然后将新任务的代码装入并运行。

图 3-17 所示演示了  $\mu\text{C}/\text{OS-II}$  操作系统中任务的创建过程，它是基于 `spawn` 模型的。

```
OS_STK TaskStk[1000];
void main(void)
{
    INT8U err;
    OSInit();
    /* OSTaskCreate: 创建一个新任务。参数：函数名，参数指针，栈顶指针，优先级 */
    OSTaskCreate(MyTask, (void *)0, &TaskStk[999], &err);
    OSStart();
}
void MyTask (void *pdata) /* 新任务的入口函数 */
{
    ...
}
```

图 3-17  $\mu\text{C}/\text{OS-II}$  操作系统中的任务创建

图 3-18 所示演示了嵌入式 Linux 操作系统中任务的创建过程,它是基于 fork/exec 模型的。

```
void main()
{
    int pid;
    pid = fork();
    if(pid > 0) /* 执行父任务代码 */
        printf("parent task");
    else if(pid == 0){ /* 执行子任务代码 */
        printf("child task");
        execvp("MyTask", NULL);
    }
}
```

图 3-18 嵌入式 Linux 操作系统中的任务创建

任务的终止可能有多种原因,例如下面三种情况。

- 正常退出: 这个任务已经完成了所有的工作,需要结束运行。在这种情况下,这个退出的要求是任务自己提出来的,所以也称为是自愿退出。
- 错误退出: 这个任务在执行过程中,由于出现了一些致命的错误,例如,执行了非法指令、出现了除 0 错误、内存访问错误等,系统就会中止该任务的运行。在这种情况下,任务不是主动退出的,而是由操作系统发现了错误以后,强制性地让它退出。
- 被其他任务踢出: 有些操作系统会提供一些系统调用函数,用来把一个任务从系统中清除出局。

在有些嵌入式系统当中,尤其是一些控制系统,它的某些任务被设计为“死循环”的模式,一直循环下去,不会终止。

### 3. 任务的状态

在多道程序系统中,任务是独立运行的实体,需要参与系统资源的竞争,只有在所需资源都得到了满足的情形下,才能在 CPU 上运行。因此,任务所拥有的资源情况是在不断变化的,这导致任务的状态也表现出不断变化的特性。不同的嵌入式操作系统对任务状态的定义不尽相同,但是一般来说,它们都会具备以下的三种基本状态。

- 运行状态 (running): 任务占有 CPU,并在 CPU 上运行。显然,处于此状态的任务个数必须小于或等于 CPU 的数目。如果在系统当中只有一个 CPU 的话,那么

在任何一个时刻，最多只能有一个任务处于运行状态。

- 就绪状态 (ready): 任务已经具备了运行的条件，但是由于 CPU 正忙，正在运行其他的任务，所以暂时不能运行。不过，只要把 CPU 分给它，它就能够立刻执行。
- 阻塞状态 (blocked): 也叫等待状态 (waiting)。任务因为正在等待某种事件的发生而暂时不能运行。例如，它正在等待某个 I/O 操作的完成，或者它同某个任务之间存在着同步关系，正在等待该任务给它发信号。此时，即使 CPU 已经空闲下来了，它也还是不能运行。

对于就绪状态和阻塞状态，它们既有相同点又有不同点。相同之处在于，任务都是处于暂停状态，没有在运行；不同之处在于，它们暂停的原因是不一样的，导致就绪状态的原因是外因，是操作系统不给 CPU 时间，而导致阻塞状态的原因是内因，是任务自身的问题。

关于任务的三个基本状态，可以用一个生活中的例子来说明。例如，假设我们的自行车坏了，需要把它送到修车铺去修理。对于这辆自行车来说，当它被交给修车师傅以后，就可能处于三种状态之一，这三种状态与任务的三种状态是一一对应的。第一种是运行状态，即修车师傅正在修理这辆自行车；第二种状态是就绪状态，也就是说，修车师傅正在忙着修理其他的自行车，而我们的自行车正在旁边等待，只要修车师傅一空闲下来，就会来修理；第三种状态是阻塞状态，例如，我们的自行车需要更换一个配件，而修车师傅那里正好没有这个配件，他已经派人去买了。这时，即使他已空闲下来，也没有办法帮我们修车。

在一定条件下，任务会在不同的状态之间来回转换，如图 3-19 所示。对于任务的三种状态，可以有四种转换关系。

- 运行→阻塞：任务由于等待某个事件而被阻塞起来。例如，一个任务正在 CPU 上运行，这时它需要用户输入一个字符。由于 CPU 的运行速度远远高于 I/O 设备的处理速度，所以操作系统不会允许该任务继续占用 CPU，在那里空等，而是把它变成阻塞状态，然后调用其他任务去运行。
- 运行→就绪：一个任务正在 CPU 上运行，这时由于种种原因（如该任务的时间片用完，或另一个高优先级任务就绪），调度器选择了另一个任务去运行。这样对于当前的任务来说，就从运行状态变成了就绪状态。
- 就绪→运行：处于就绪状态的任务被调度器选中去运行。
- 阻塞→就绪：一个任务曾经因为等待某个事件而被阻塞起来，如果它等待的事件发生了，那么该任务就从阻塞状态变成了就绪状态，从而具备了继续运行的条件。

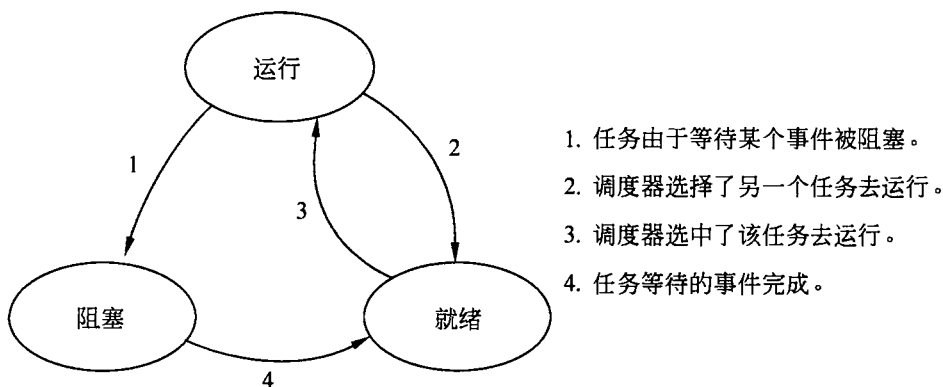


图 3-19 任务间的状态转换图

由于在系统当中，同时存在着多个任务，因此，从宏观上看，好像这些任务都处于活动状态。但实际上，对于每一个任务来说，在它的生命周期的每个时刻，都是在这三种状态之间来回转换。如果需要 I/O 操作，就进入阻塞状态；如果没有涉及到 I/O 操作，那么它就不停地在运行和就绪这两个状态之间来回地转换，不断地循环。这样，就使得系统当中的每一个任务都有机会得到 CPU 去运行。另外，这种转换完全是自动进行的，是由操作系统来完成的，对于用户和进程本身来说，是意识不到的。

#### 4. 任务控制块

从操作系统的角度来说，它是如何来设计和实现任务机制的呢？是如何来进行任务管理的呢？答案就是任务控制块（Task Control Block, TCB），任务管理就是通过对各个任务的 TCB 的操作来实现的。

所谓 TCB，就是在操作系统当中，用来描述和管理一个任务的数据结构。系统为每一个任务都维护了一个相应的 TCB，用来保存该任务的各种相关信息。TCB 的内容主要包括以下几项。

- 任务的管理信息：包括任务的标识 ID、任务的状态、任务的优先级、任务的调度信息、任务的时间统计信息、各种队列指针等。
- CPU 上下文信息：各种 CPU 寄存器的当前值，包括通用寄存器、PC 寄存器、程序状态字、栈指针等。前面在讨论进程的特性时，曾经谈到，物理上的硬件寄存器只有一份，但每个进程都有一份各自独立的逻辑寄存器。所谓的逻辑寄存器，实际上就是 TCB 当中的相应字段，是一些内存变量。另外，在实际的嵌入式系统中，CPU 上下文信息不一定直接存放在 TCB 当中，而是存放在任务的栈当中，可以通过相应的栈指针来访问。

- 资源管理信息：如果在操作系统中，任务表示的是进程，则还应包含一些资源管理方面的信息，如段表地址、页表地址等存储管理方面的信息，根目录、文件描述字等文件管理方面的信息。

有了 TCB 这个数据结构以后，任务的管理就比较具体、落实了。我们可以用 TCB 来描述任务的基本情况以及它的运行变化过程，把 TCB 看成是任务存在的唯一标志。具体来说，当需要创建一个新任务的时候，就为它生成一个 TCB，并初始化这个 TCB 的内容；当需要终止一个任务的时候，只要回收它的 TCB 就可以了。而对于任务的组织和管理，也可以通过它们的 TCB 的组织和管理来实现。

### 5. 任务切换

假设一个任务正在 CPU 上运行，这时由于某种原因，系统决定调度另一个任务去运行，那么在这种情形下，就要进行一次任务切换（context switching），把当前任务的运行上下文保存起来，并恢复新任务的上下文。

任务切换通常具有如下基本步骤：

- (1) 将处理器的运行上下文保存在当前任务的 TCB 中。
- (2) 更新当前任务的状态，从运行状态变为就绪状态或阻塞状态。
- (3) 按照一定的策略，从所有处于就绪状态的任务中选择一个去运行。
- (4) 修改新任务的状态，从就绪状态变成运行状态。
- (5) 根据新任务的 TCB 的内容，恢复它的运行上下文环境。

图 3-20 所示是通过一个例子演示任务切换的过程。假设在系统中只有两个任务，图的左边是任务 1 的运行情况，图的右边是任务 2 的运行情况，图的中间是操作系统所做的事情，时间轴的方向为从上往下。在刚开始时，任务 1 处于运行状态。

(1) 当任务 1 运行了一段时间后，系统中发生了一个中断或者该任务进行了一次系统调用，这样，控制流就进入了操作系统，开始执行操作系统的代码，以完成此次中断处理或系统调用服务。

(2) 操作系统在执行过程中，基于某种原因，发现任务 1 已经不具备在 CPU 上运行的资格。例如，刚刚发生的是一次系统调用，任务 1 需要去访问 I/O 设备，等待用户的键盘输入；或者刚刚发生的是一个时钟中断，在中断处理中，系统发现任务 1 的时间片已用完；或者刚刚发生的是一次 I/O 中断，表明某个 I/O 操作已经完成，所以系统把相应的任务变为就绪状态，而该任务的优先级要高于当前任务。总之，基于某种原因，系统认为当前任务已经不适合继续在 CPU 上运行。

(3) 操作系统决定进行一次任务切换。首先把任务 1 的所有状态信息都保存起来，包括 CPU 当中各个寄存器的值（PC 寄存器、PSW 寄存器、通用寄存器、段寄存器、栈指针寄存器等），把它们保存在任务控制块 TCB 当中，或者是保存在任务的栈当中，并将栈顶

地址保存在 TCB 中。同时,任务的状态也发生了变化,从原来的运行状态变成就绪状态或阻塞状态。

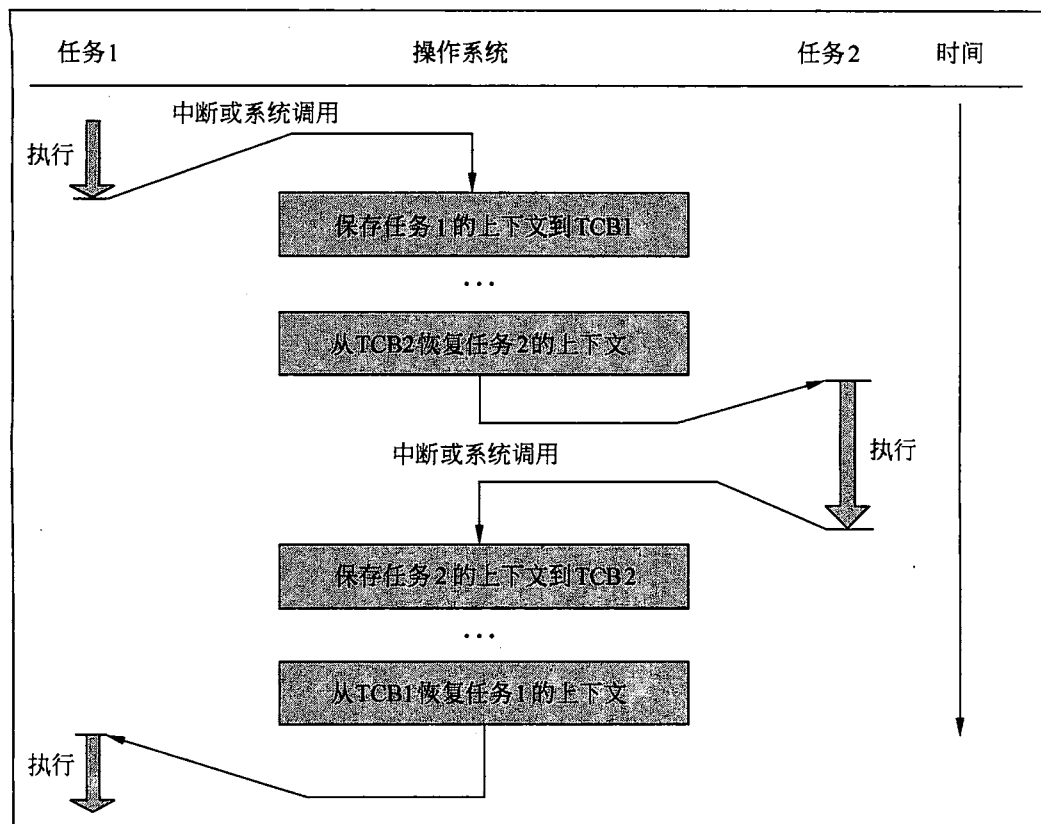


图 3-20 任务切换示意图

(4) 操作系统会从所有处于就绪状态的任务当中,选择一个去运行,假设它选择了任务 2。这时,操作系统就会从任务 2 的 TCB 当中,把它的所有状态信息都读取出来,并装入到相应的硬件寄存器当中,从而恢复该任务在上次运行中止时的现场,重新开始运行。同时,任务的状态也从原来的就绪状态转换成运行状态。

(5) 任务 2 在执行一段时间后,也出现了一次中断或系统调用。基于类似的原因,操作系统又决定进行一次任务切换(并非每次中断或系统调用都会引起任务切换),这样,操作系统就把任务 2 的所有状态信息保存在它的 TCB 当中,并且根据任务 1 的 TCB 的内容,恢复它的运行上下文,使它重新开始运行。

(6) 在整个过程中, 开始是任务 1 在运行, 然后是操作系统在运行, 然后是任务 2 在运行, 接下来又是操作系统在运行, 最后是任务 1 在运行。在这个过程当中, 任务的状态在不断发生变化。

## 6. 任务队列

如前所述, 在一个多任务的操作系统当中, 各个任务的状态是经常变化的, 有时处于运行状态, 有时处于就绪状态, 有时又处于阻塞状态。即便同是阻塞状态, 引发阻塞的原因可能又各不相同, 有的是因为等待 I/O 操作, 有的是因为任务之间的同步。因此, 在一个操作系统当中, 采用什么样的方式来组织它的所有任务, 将直接影响到对这些任务的管理效率。

通常的做法是采用任务队列的方式。也就是说, 由操作系统来维护一组队列, 用来表示系统当中所有任务的当前状态。不同的状态用不同的队列来表示。例如, 处于运行状态的所有任务构成了运行队列, 处于就绪状态的所有任务构成了就绪队列, 而对于处于阻塞状态的任务, 则要根据它们阻塞的原因, 分别构成相应的阻塞队列。然后, 对于系统当中的每一个任务, 根据它的状态把它的 TCB 加入到相应的队列当中去。如果一个任务的状态发生变化, 例如, 从运行状态变成就绪状态, 或者从阻塞状态变成就绪状态, 这时, 就要把它的 TCB 从一个状态队列中脱离出来, 加入到另一个队列当中去。

图 3-21 所示是任务队列的一个例子, 包含了就绪队列和各种阻塞队列。在操作系统当中, 就绪队列只有一个, 所有处于就绪状态的任务, 它们的 TCB 都插入在该队列中。在本例中, 就绪队列包含有三个节点, 这说明在当前时刻, 有三个任务处于就绪状态。接下来有三个阻塞队列: 第一个是信号量 S 的阻塞队列, 所有正在等待该信号量的任务, 都挂在这个队列上, 目前有两个这样的任务; 第二个是邮箱 M 的阻塞队列, 用来描述正在等待邮箱 M 的任务; 第三个是磁盘访问队列, 目前该队列是空的, 表示没有任务正在进行磁盘访问操作。总之, 针对不同的阻塞原因, 系统会构造不同的阻塞队列。假设在某个时刻, 信号量 S 被释放, 则任务 TCB5 会被唤醒, 它的状态就会从阻塞状态变成就绪状态, 所以我们要把 TCB5 从阻塞队列里摘下来, 并把它加入到就绪队列当中。另外, 图 3-21 中没有把运行队列包含进来, 这是因为对于单 CPU 的嵌入式系统而言, 在任何一个时刻, 只可能有一个任务在 CPU 上运行, 所以在运行队列当中, 始终只有一个任务。

### 3.3.4 任务的调度

#### 1. 任务调度概述

在多道程序操作系统中, 经常会出现多个任务同时去竞争 CPU 的情况。换句话说, 就是在系统的就绪队列中, 有两个或多个任务同时处于就绪状态。假设在系统中只有一个



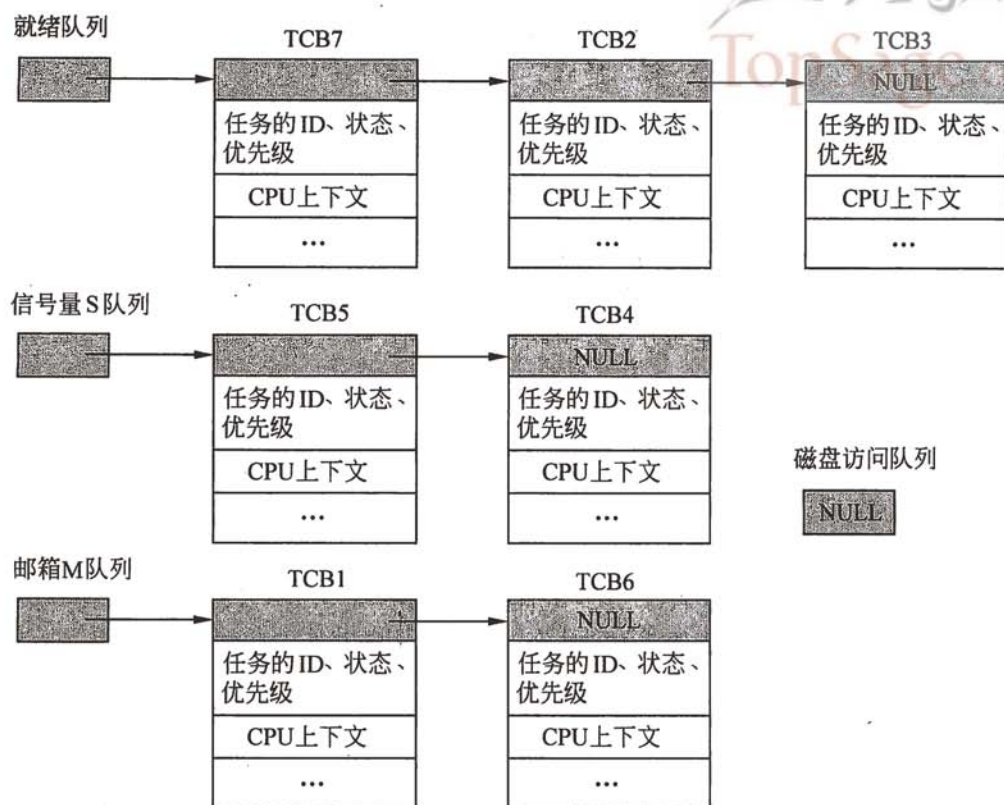


图 3-21 任务队列

CPU，而且这个 CPU 已经空闲下来了，现在的问题就是：对于就绪队列当中的那些任务，应该选择哪一个去运行？在操作系统当中，负责去做出这个选择的那一部分程序，就称为是调度器 (scheduler)，而调度器在决策过程中所采用的算法，就称为是调度算法。从资源管理的角度来看，也可以把调度器看成是 CPU 这个资源的管理者。

任务调度的首要问题是何时进行调度，即调度发生的时机。一般来说，在以下几种情形下，可能会发生任务的调度。

- 当一个新的任务被创建时，需要做出一个调度决策，是立即执行这个新任务还是继续执行父任务？
- 当一个任务运行结束时，它不再占用 CPU，这时调度器必须做出一个决策，从就绪队列中选择某个任务去运行。如果此时没有任务处于就绪状态，系统一般会调度一个特殊的空闲任务。
- 当一个任务由于 I/O 操作、信号量或其他原因被阻塞时，也必须另选一个任务

运行。

- 当一个 I/O 中断发生时, 表明某个 I/O 操作已经完成, 而等待该 I/O 操作的任务将从阻塞状态变为就绪状态, 此时可能需要做出一个调度决策, 是立即执行这个新就绪的任务, 还是继续执行刚才被中断的那个任务。
- 当一个时钟中断发生时, 表明一个时钟节拍已经结束, 这时, 可能会唤醒一些延时的任务, 使它们变为就绪状态。也可能会发现当前任务的时间片已用完, 从而把它变为就绪状态。在这些情况下, 也需要调度器来重新调度。

任务调度的第二个问题是调度的方式, 主要有两种方式: 可抢占调度和不可抢占调度。

- 不可抢占方式 (nonpreemptive): 如果一个任务被调度程序选中, 就会一直地运行下去, 直到它因为某种原因 (如 I/O 操作或任务间的同步) 被阻塞了, 或者它主动地交出了 CPU 的使用权。换句话说, 如果有一个任务长时间地占用着 CPU, 系统也不会强制它中止。在不可抢占的调度方式下, 当出现调度时机当中的前三种情形时, 即新任务创建、任务运行结束及任务被阻塞, 都有可能发生调度。而对于第四种和第五种情形, 即发生各种中断的时候, 虽然也会有中断处理程序, 但是它并不会去调用调度程序。因此, 当中断处理完成后, 又会回到刚才被打断的任务中继续执行。
- 可抢占方式 (preemptive): 当一个任务正在运行的时候, 调度程序可以去打断它, 并安排另外的任务去运行。在这种调度方式下, 对于调度时机当中的所有五种情况, 都有可能发生调度。另外, 在其他的一些情况下, 例如, 假设调度算法是按照任务的优先级来进行调度, 那么一旦在就绪队列当中有任务的优先级高于当前正在运行的任务, 就可能立即进行调度, 转让 CPU。

实时操作系统大都采用了可抢占的调度方式, 使一些比较重要的关键任务能够打断那些不太重要的非关键任务的执行, 以确保关键任务的截止时间能够得到满足。

在嵌入式操作系统当中, 存在着许多的调度算法, 每一种算法都有各自的优点和缺点。因此, 任务调度的第三个问题是调度算法的性能指标, 即如何来评价一个调度算法的好坏。这些指标主要包括下面几项。

- 响应时间 (response time): 调度器为一个就绪任务进行上下文切换时所需的时间, 以及任务在就绪队列中的等待时间。
- 周转时间 (turnaround time): 一个任务从提交到完成所经历的时间。
- 调度开销 (overhead): 调度器在做出调度决策时所需要的时间和空间开销。
- 公平性 (fairness): 大致相当的两个任务所得到的 CPU 时间也应该是大致相同的。另外, 要防止饥饿 (starvation), 即某一个任务始终得不到处理器去运行。
- 均衡性 (balance): 要尽可能使整个系统的各个部分 (CPU、I/O) 都忙起来, 提

高系统资源的使用效率。

- 吞吐量 (throughput): 单位时间内完成的任务数量。

在这些指标当中,有一些是可以共存的,也有一些是相互牵制的。因此,对于一个实际的调度算法来说,这些指标不可能全部都实现,而是要根据系统的需要,有一个综合的权衡和折中的过程。

## 2. 先来先服务算法

最简单的一种调度算法是先来先服务算法 (First Come First Served, FCFS), 也叫做先进先出算法 (First In First Out, FIFO)。顾名思义, 先来先服务的基本思想就是按照任务到达的先后次序来进行调度, 如图 3-22 所示。它是一种不可抢占的调度方式, 如果当前任务占用着 CPU 在运行, 那么就要一直等到它执行完毕或者因为某种原因被阻塞, 才会让出 CPU 给其他的任务。另外, 对于一个被阻塞的任务, 当它被唤醒之后, 就会把它放在就绪队列的末尾, 重新开始排队。

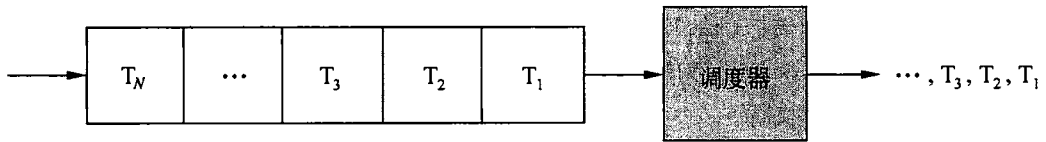


图 3-22 FCFS 算法示意图

先来先服务算法的最大优点就是简单, 易于理解也易于实现。它的缺点也很明显: 一批任务的平均周转时间取决于各个任务到达的顺序, 如果短任务位于长任务之后, 那么将增大平均周转时间。例如, 假设有三个任务 A、B、C, 它们实际需要的运行时间分别是 24、3 和 3, 单位是分钟。这三个任务几乎同时到达, 但是 A 稍微早一点, B 和 C 稍微晚一点, 按照先来先服务的原则, A 先执行, 然后是 B 和 C, 如图 3-23(a)所示。因此, 在前 24 分钟内是 A 在执行, 接下来的 3 分钟是 B 在执行, 最后的 3 分钟是 C 在执行。由于 A 的到达时间是 0, 结束时间是 24, 所以周转时间是 24。B 的到达时间是 0, 结束时间是 27, 所以周转时间是 27。同样, C 的周转时间是 30。因此在这种情况下, 这三个任务的平均周转时间是 27 分钟。假设我们把这三个任务的到达顺序调整一下, B 先到, 然后是 C 和 A。在这种情况下, 前三分钟是执行 B, 接下来的 3 分钟是执行 C, 再接下来的 24 分钟是执行 A, 如图 3-23(b)所示。这样的话, B 的周转时间是 3, C 的周转时间是 6, 而 A 的周转时间是 30, 最后的平均周转时间是 13 分钟。显然, 在这种情况下, 平均周转时间得到了大幅度的降低, 减少了一半左右。两者唯一的区别, 仅仅是把两个任务的执行顺序调整了一下, 由此可见调度算法的重要性。

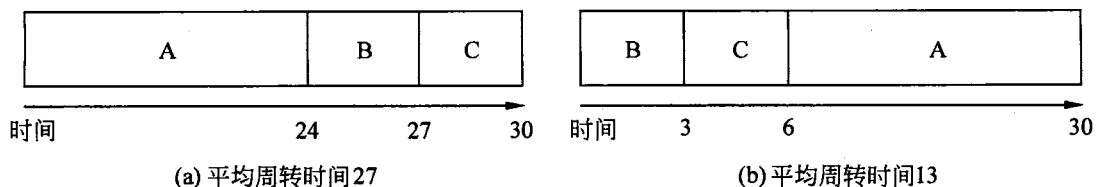


图 3-23 FCFS 算法的例子

### 3. 短作业优先算法

为了改进 FCFS 算法,减少平均周转时间,人们又提出了短作业优先算法 (Shortest Job First, SJF)。SJF 算法的基本思路是:各个任务在开始执行前,必须事先预计好它的执行时间,然后调度算法将根据这些预计时间,从中选择用时较短的任务优先执行。SJF 算法有两种实现方案。

- 不可抢占方式:当前任务正在运行的时候,即使来了一个比它更短的任务,也不会被打断,只有当它运行完毕或者是被阻塞时,才会让出 CPU,进行新的调度。
- 可抢占方式:如果一个新的短任务到来了,而且它的运行时间要小于当前正在运行的任务的剩余时间,那么这个新任务就会抢占 CPU 去运行。这种方法,也称为最短剩余时间优先算法 (Shortest Remaining Time First, SRTF)。

不可抢占的 SJF 算法如图 3-24 所示,由于任务  $T_3$  的执行时间最短,所以首先被调度运行,其次是  $T_1$  和  $T_2$ 。

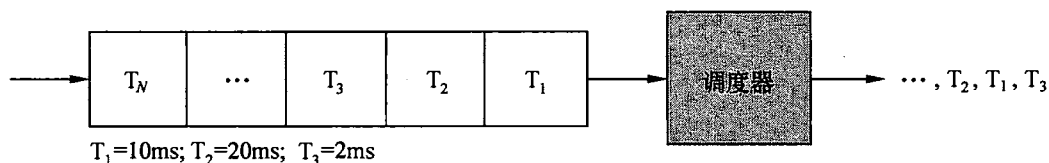


图 3-24 SJF 算法示意图

可以证明,对于一批同时到达的任务,采用 SJF 算法将得到一个最小的平均周转时间。例如,假设有四个任务 A、B、C、D,它们的运行时间分别是  $a$ 、 $b$ 、 $c$  和  $d$ ,假设它们的到达时间是差不多的,调度顺序为 A、B、C、D。那么任务 A 的周转时间为  $a$ ,B 的周转时间为  $a+b$ ,C 的周转时间为  $a+b+c$ ,D 的周转时间为  $a+b+c+d$ ,因此,最后的平均周转时间为  $(4a+3b+2c+d)/4$ 。从这个式子来看,显然,只有当  $a \leq b \leq c \leq d$  的时候,这个平均周转时间才会达到一个最小值。这个结论可以推广到任意多个任务的情况下。

### 4. 时间片轮转算法

时间片轮转算法 (Round Robin, RR) 的基本思路是:把系统当中的所有就绪任务按

照先来先服务的原则，排成一个队列，然后，在每次调度的时候，把处理器分派给队列当中的第一个任务，让它去执行一小段 CPU 时间，或者叫时间片（time slice）。当这个时间片结束的时候，如果任务还没有执行完的话，将会发生时钟中断，在时钟中断里面，调度器将会暂停当前任务的执行，并把它送到就绪队列的末尾，然后执行当前的队首任务。反之，如果一个任务在它的时间片用完之前就已经运行结束了或者是被阻塞了，那么它就会立即让出 CPU 给其他的任务。

图 3-25 所示是时间片轮转法的一个示意图。图 3-25(a)表示初始状态，总共有四个任务位于就绪队列中，它们的先后顺序分别是 B、C、D、A，其中任务 B 位于队列之首。因此，当 CPU 空闲时，调度器就会选择 B 去运行。假设当它运行完一个时间片以后，既没有运行结束，也没有被阻塞，这时操作系统就会通过时钟中断来中止它的运行，并把它送到就绪队列的末尾，也就是图 3-25(b)的状态。此时，任务 C 位于队列之首，所以它被调度执行。同样，当它的时间片用完之后，也会被送到就绪队列的末尾，然后任务 D 又可以开始运行。就这样不断地循环往复，使得每个任务都能够轮流地执行一段时间，直到某个时候，有一个任务运行结束了，它就退出这个就绪队列。或者，如果该任务在运行的时候，由于 I/O 操作等原因被阻塞了，那么也会退出这个就绪队列，并加入到相应的阻塞队列中去。

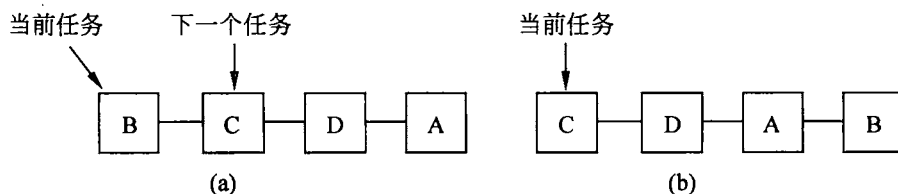


图 3-25 RR 算法示意图

时间片轮转法的优点如下。

- 公平性：各个就绪任务平均地分配 CPU 的使用时间。例如，假设有  $n$  个就绪任务，那么每个任务将得到  $1/n$  的 CPU 时间。
- 活动性：每个就绪任务都能一直保持着活动性。假设时间片的大小为  $q$ ，那么每个任务最多等待  $(n-1)q$  长的时间，就能再次得到 CPU 去运行。

在采用时间片轮转算法时，时间片的大小  $q$  要适当选择，如果选择不当，将影响到系统的性能和效率。

- 如果  $q$  太大，每个任务都在一个时间片内完成，这就失去了轮转法的意义，退化为先来先服务算法了，从而使各个任务的响应时间变长。例如，如果把  $q$  设置为

100ms, 而且在就绪队列中有 10 个任务, 那么对于排在队列末尾的那个任务, 可能要等上近 1 秒钟才能够得到 CPU 去运行。

- 如果  $q$  太小, 每个任务就需要更多的时间片才能运行完, 这就使任务之间的切换次数增加, 从而增大了系统的管理开销, 降低了 CPU 的使用效率。例如, 假设把  $q$  设置为 4ms, 而每次任务切换的时间开销是 1ms, 这就意味着 CPU 每工作 4ms 的时间, 就必须花 1ms 来进行任务切换, 即 CPU 的利用率只有 80%, 这显然是不能接受的。

因此, 选择一个合适的  $q$  值, 既不能太大, 也不能太小, 这是时间片轮转法的最大问题。一般来说, 这个值选在 20ms~50ms 之间是比较合适的。

### 5. 优先级算法

时间片轮转法有一个默认的前提, 即位于就绪队列当中的各个任务是同等重要的, 所以把它们按照先来后到的顺序排成一个队列, 大家轮流执行相同长度的时间片。但是在实际的系统当中, 尤其是在一些嵌入式实时操作系统当中, 并不是每个任务都是同等重要的, 不同的任务对响应时间的要求是不一样的, 所以对它们的处理也应该有所区别。

优先级调度算法 (priority) 的基本思路是: 给每一个任务都设置一个优先级, 然后在任务调度的时候, 在所有处于就绪状态的任务中选择优先级最高的那个任务去运行。例如, 短作业优先算法其实也是一个优先级算法, 每个任务的优先级就是它的运行时间, 运行时间越短, 优先级越高。

优先级算法可以分为两种: 可抢占方式和不可抢占方式。它们的区别在于: 当一个任务正在运行的时候, 如果这时来了一个新的任务, 其优先级更高, 那么在这种情况下, 是立即抢占 CPU 去运行新任务, 还是等当前任务运行完后再决定。

图 3-26 所示是可抢占优先级调度算法的一个例子, 它有三个任务, 任务 1 的优先级最低, 任务 3 的优先级最高。在刚开始时, 任务 1 在 CPU 上运行, 这时任务 2 进入就绪状态, 由于它的优先级更高, 所以就立即抢占了 CPU 去运行, 而任务 1 从运行状态变成了就绪状态。当任务 2 运行一段时间后, 任务 3 也进入了就绪状态, 所以它又打断了任务 2 的运行。当任务 3 执行完毕后, 此时在就绪队列中有两个任务, 所以选择了较高优先级的任务 2 去运行, 当它也运行完毕后, 才轮到任务 1 去运行。

在任务优先级的确定方式上, 可以分为静态方式和动态方式两种。

- 静态优先级方式: 在创建任务的时候就确定任务的优先级, 并且一直保持不变直到任务结束。优先级的确定可以依据任务的类型或重要性, 例如, 系统任务的优先级要高于用户任务, 实时任务的优先级要高于非实时任务。静态优先级方式有一个很大的缺点: 高优先级的任务会一直占用着 CPU 运行, 而那些低优先级的任务可能会长时间地得不到 CPU, 一直处于“饥饿”状态。

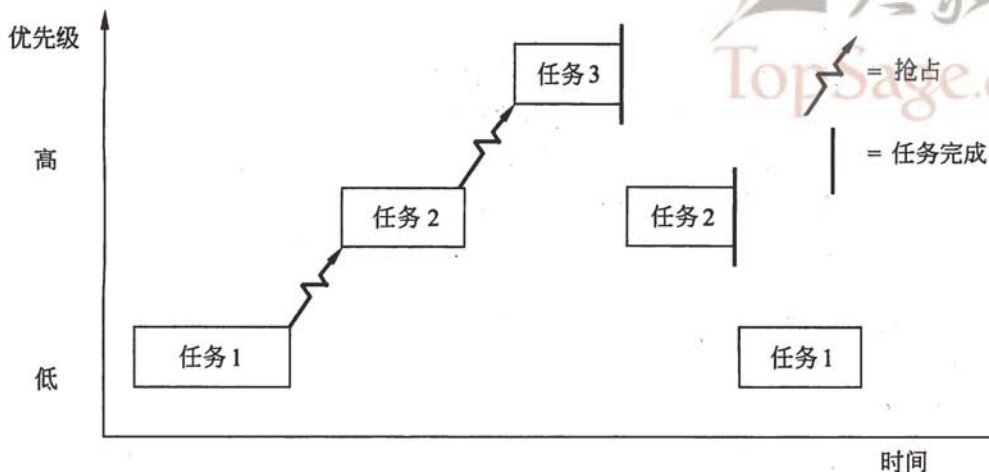


图 3-26 可抢占的优先级算法

- 动态优先级方式：在创建任务的时候确定任务的优先级，但是该优先级可以在任务的运行过程中动态改变，以便获得更好的调度性能。例如，为了防止静态优先级方式中出现的“饥饿”现象，系统可以根据任务占用 CPU 的运行时间和它在就绪队列中的等待时间来不断地调整它的优先级。这样，对于一个优先级比较低的任务，如果它在就绪队列中的等待时间足够长，那么它的优先级就会不断提高，最终可以被调度执行。

在优先级算法中，高优先级的任务将抢占低优先级的任务，但如果两个任务的优先级相同，又该如何处理呢？通常的做法是把任务按照不同的优先级进行分组，然后在不同组的任务之间使用优先级算法，而在同一组的各个任务之间使用时间片轮转法。例如，在图 3-27 中，系统将优先级划分为四个类别，优先级 4 表示最高优先级，优先级 1 表示最低优先级，每一个优先级都有一个相应的就绪队列。然后对于系统当中的每一个就绪任务，根据它的优先级把它加入到相应的就绪队列当中。如图 3-27 所示，在优先级 4 的就绪队列中，有三个就绪任务。在优先级 3 和 2 的就绪队列中，分别挂了一个和四个任务，而优先级 1 的就绪队列是空的。对于这样的一个系统，调度算法的工作原理是这样的：只要在优先级 4 的就绪队列中存在有就绪任务，那就对这些任务实行时间片轮转法，一个接一个地轮流运行一个时间片，直到这些任务都已经运行完毕或被阻塞起来，即相应的就绪队列为空。然后，再去考虑比它低一级的就绪队列，也采用相同的处理方法。就这样一级一级下来，先执行高优先级的任务，再执行低优先级的任务，而对于相同优先级的任务，则使用时间片轮转法。当然，为了避免低优先级的任务始终得不到运行机会，处于“饥饿”状态，也可以动态调整任务的优先级。



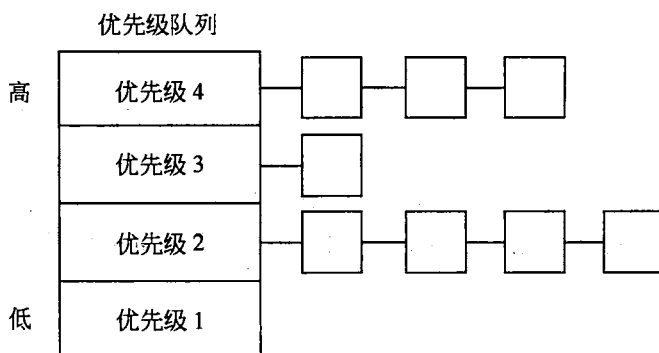


图 4.27 优先级队列

采用优先级调度算法，还有一个问题就是可能会发生优先级反转（priority inversion）的现象。在理想情况下，当高优先级任务处于就绪状态后，会立即抢占低优先级任务而得到执行。但在实际系统当中，在各个任务之间往往需要用到各种共享资源，如 I/O 设备、信号量、邮箱等。在这种情况下，可能会出现高优先级任务被低优先级任务阻塞，等待它释放资源，而低优先级任务又在等待中等优先级任务的现象，这种现象称为“优先级反转”。

图 3-28 所示是一个优先级反转的例子。假设在刚开始的时候，有一个优先级为 4 的任务 A 正在运行。当它运行到  $t_1$  时刻的时候，它申请并得到了共享资源 R，然后继续运行。在  $t_2$  时刻，有一个优先级为 11 的任务 B 就绪了，由于它的优先级比 A 要高，所以就抢占了 CPU 去运行，而任务 A 就从运行状态变成就绪状态。在  $t_3$  时刻，任务 B 也需要用到资

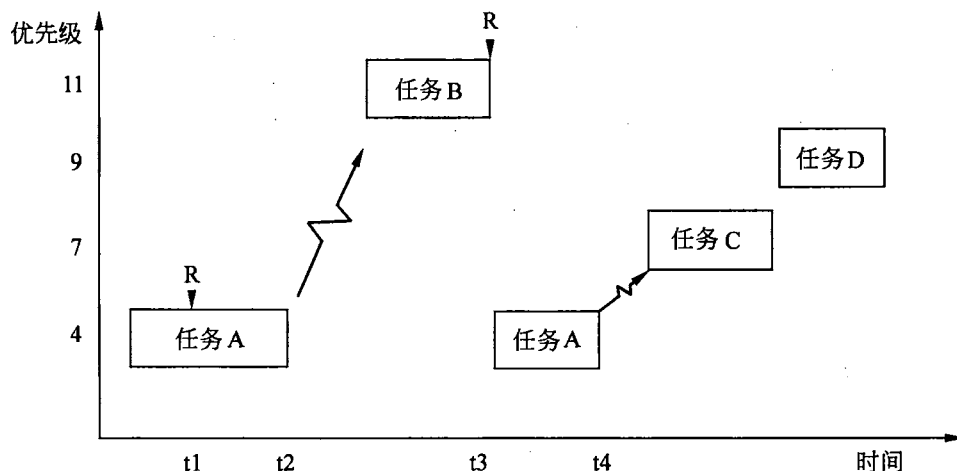


图 3-28 优先级反转



源 R, 但由于 R 已经被 A 占用了, 所以任务 B 进入了阻塞状态, 等待该资源被释放。这时, 任务 A 是唯一处于就绪状态的任务, 所以它重新开始执行, 但是它还没有来得及释放资源 R, 在  $t_4$  时刻, 另一个优先级为 7 的任务 C 就绪了, 由于它的优先级要高于 A, 所以立即被调度执行。当它运行完后, 又来了一个优先级为 9 的任务 D, 它又开始运行。就这样, 由于任务 A 的优先级比较低, 所以在随后的一段时间内, 只要有优先级更高的任务就绪, 它就始终无法运行, 处于“饥饿”状态。同时, 由于它无法释放资源 R, 这使得高优先级的 B 也无法执行。

### 3.3.5 实时系统调度

许多嵌入式操作系统都是实时操作系统, 对于 RTOS 调度器来说, 任务之间的公平性并不是最重要的, 它追求的是实时性, 即要让每个任务都在其最终时间期限(deadline)之前完成。

大多数 RTOS 调度器都采用了基于优先级的可抢占调度算法, 但是在具体实现上, 需要考虑几个方面的问题, 例如, 如何设定各个任务的优先级? 优先级是静态设置的还是动态可变的? 算法的性能如何, 能否满足实时要求?

#### 1. 任务模型

考虑实时系统中常用的任务模型, 即周期性任务模型。所谓的周期任务, 就是该任务每隔固定的一段时间, 就会运行一次。

首先定义如下的数据结构。

- 启动时间  $r(i, j)$ : 第  $i$  个任务的第  $j$  次执行的开始时间。
- 时间期限 (deadline)  $D(i)$ : 第  $i$  个任务所允许的最大响应时间 (从任务启动到运行结束所需的时间)。
- 周期 (period)  $P(i)$ : 第  $i$  个任务的连续两次运行之间的最小时间间隔;
- 执行时间 (execution time)  $E(i)$ : 对于第  $i$  个任务, 当它所需要的资源都已具备时, 它的执行所需要的最长时间。

在任务模型中, 每一个任务用一个三元组来表示 (执行时间、周期、deadline)。一般来说, 一个任务的周期时间同时也是它的 deadline, 因为该任务必须在它的下一个周期开始之前, 完成此次运行。另外, 任务可以在一个周期内的任何时刻被启动, 但必须在它的时间期限之前完成, 如图 3-29 所示。

#### 2. RMS 算法

单调速率调度算法 (Rate Monotonic Scheduling, RMS) 是一种静态优先级调度算法, 也是最常用的一种确定任务优先级的算法。RMS 算法基于以下几个假设条件。

- 所有的任务都是周期性任务。

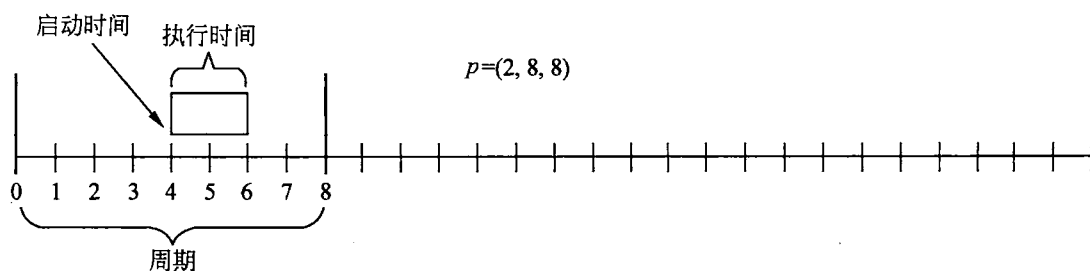


图 3-29 RTOS 任务模型

- 任务的时间期限等于它的周期。
- 任务在每个周期内的执行时间是一个常量。
- 任务之间不进行通信，也不需要同步。
- 任务可以在任何位置被抢占，不存在临界区的问题。

RMS 算法的基本思路：任务的优先级与它的周期表现为单调函数的关系，任务的周期越短，优先级越高；任务的周期越长，优先级越低。

RMS 算法是一种最优调度算法：如果存在一种基于静态优先级的调度顺序，使得每个任务都能在其期限内完成，那么 RMS 算法总能找到这样的一种可行的调度方案。当然，对于具体的某一组任务而言，这种调度方案并不一定存在。但只要存在，就能通过 RMS 算法进行调度。

为了判断一组任务的可调度性，可以计算 CPU 的使用率： $U = \sum_i \frac{E_i}{P_i}$ 。如前所述， $E_i$

是第  $i$  个任务的执行时间， $P_i$  是它的周期。

- 如果  $U > 1$ ，则 RMS 调度方案不存在（处理器不可能一天工作 25 个小时）。
- 如果  $U \leq n(2^{1/n} - 1)$ ， $n$  为任务的个数，则 RMS 调度方案一定存在。
- 如果  $n(2^{1/n} - 1) < U \leq 1$ ，则 RMS 调度方案可能存在也可能不存在。

令  $T = n(2^{1/n} - 1)$ ，表示可调度上限。例如，当  $n = 1$  时， $T = 1$ ；当  $n = 2$  时， $T = 0.83$ ；当  $n$  趋向于无穷大时， $T = \ln 2 = 0.69$ 。

例如，如表 3-1 所示，有两个任务  $T_1$  和  $T_2$ 。 $T_1$  的执行时间为 2，周期和时间期限为 5； $T_2$  的执行时间为 4，周期和时间期限为 7。由于  $T_1$  的周期更短，所以它的优先级要高于  $T_2$ 。另外，在该系统当中，CPU 的使用率  $U = 2/5 + 4/7 = 0.97$ ，而 RMS 的可调度上限  $T = 0.83$ ，因此，在这种情形下，RMS 无法保证能够找到合适的调度顺序，使得每个任务都能在自己的时间期限之前完成。

表 3-1 RMS 举例

任 务	执行时间	周 期	deadline
T <sub>1</sub>	2	5	5
T <sub>2</sub>	4	7	7

图 3-30 所示是 RMS 算法超时的一个例子。由于 T<sub>1</sub> 的优先级要高于 T<sub>2</sub>，所以每次当它处于就绪状态时，都能立即从 T<sub>2</sub> 中抢占 CPU 去运行。因此，T<sub>1</sub> 的运行情况非常稳定，即每隔 5 个时间单位，T<sub>1</sub> 就会去运行，而且当它在运行的时候，不会被打断。相反，任务 T<sub>2</sub> 的运行需要考虑两个方面的问题：一方面，它必须在每个周期内运行一次；另一方面，它只能在 T<sub>1</sub> 不运行的时候才能运行。

具体来说，在  $t_0$  时刻，任务 T<sub>1</sub> 和 T<sub>2</sub> 都想运行，但 T<sub>1</sub> 的优先级更高，所以它首先抢占了 CPU。在  $t_2$  时刻，T<sub>1</sub> 运行结束，T<sub>2</sub> 开始运行。它原本需要运行 4 个时间单位，但是在  $t_5$  时刻，任务 T<sub>1</sub> 又开始新一轮运行，直到  $t_7$  结束。而对于任务 T<sub>2</sub> 而言，它在上一轮运行中还剩下 1 个时间单位，而新的一个周期又开始了，这样，就造成了它的一次超时，没有满足规定的时间期限。在  $t_7$  时刻，任务 T<sub>2</sub> 首先把上一轮的 1 个时间单位运行完，然后又开始新一轮运行，但是在  $t_{10}$  时刻，又被任务 T<sub>1</sub> 打断，并在  $t_{12}$  时刻重新恢复运行，这一次它赶在了  $t_{14}$ ，也就是它的第二个时间期限之前完成，所以没有超时。从  $t_{14}$  时刻开始，任务 T<sub>1</sub> 和 T<sub>2</sub> 轮流使用 CPU，再也没有发生超时的现象。

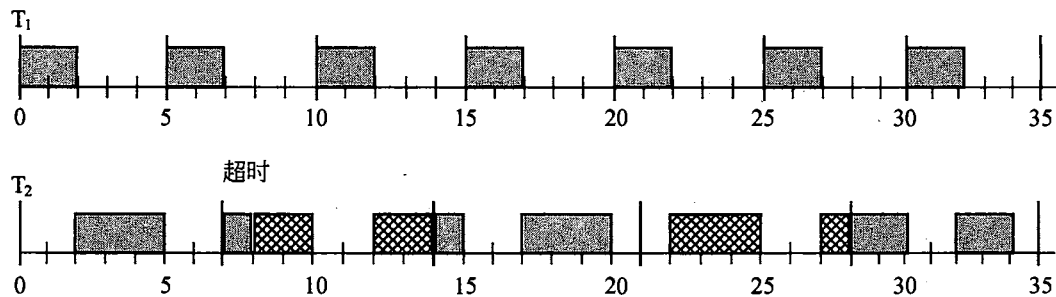


图 3-30 RMS 算法超时的例子

在任务比较多的情况下，RMS 可调度的 CPU 使用率上限为  $\ln 2 = 0.69$ ，如此低的 CPU 使用率对于大多数系统来说是不可接受的。另外，RMS 的不足之处还在于它假定任务是相同独立的、周期性的，且任务能够在任何位置被抢占，而在一个实际的系统中，任务之间通常都需要进行通信和同步。

### 3. EDF 算法

最早期限优先 (Earliest Deadline First, EDF) 调度算法是一种动态优先级调度算法，

它能根据需要动态地修改各个任务的优先级，是目前性能较好的一种调度算法。

EDF 算法的基本思路是：根据任务的截止时间来确定其优先级，对于时间期限最近的任務，分配最高的优先级。当有一个新的任务处于就绪状态时，各个任务的优先级就有可能要进行调整。与 RMS 算法一样，EDF 算法的分析也是在一系列假设的基础上进行的，除了它不要求系统中的任务都必须是周期任务，其他的假设条件与 RMS 相同。

EDF 算法是最优的单处理器动态调度算法，其可调度上限为 100%。对于给定的一组任务，只要它们的 CPU 使用率小于或等于 1，EDF 就能找到合适的调度顺序，使得每个任务都能在自己的时间期限内完成。反之，如果 EDF 不能满足这组任务的调度要求，则其他的调度算法也不行。

仍以表 3-1 中的系统为例，在 RMS 方式下，任务  $T_2$  会发生超时的现象。但如果采用 EDF 算法，则可以避免这个问题。

如图 3-31 所示，在  $t_0$  时刻，任务  $T_1$  和  $T_2$  都进入了就绪状态，想要运行。由于  $T_1$  的截止时间是  $t_5$ ， $T_2$  的截止时间是  $t_7$ ，显然任务  $T_1$  更为紧迫，所以它的优先级更高，被首先调度执行。在  $t_2$  时刻， $T_1$  运行结束， $T_2$  开始运行。到了  $t_5$  时刻，任务  $T_1$  进入了一个新的周期，它又想运行。但是在当前状态下， $T_1$  的截止时间是  $t_{10}$ ，而  $T_2$  仍然处于第一个运行周期，它的截止时间还是  $t_7$ ，显然  $T_2$  的优先级更高，所以它继续运行。而  $T_1$  在等待了一个时间单元后，在  $t_6$  时刻开始运行。这样，任务  $T_1$  和  $T_2$  在它们的第一个运行周期内，都没有发生超时的现象，这一点与刚才的 RMS 算法是不同的。在 RMS 算法中，任务  $T_2$  发生了超时。

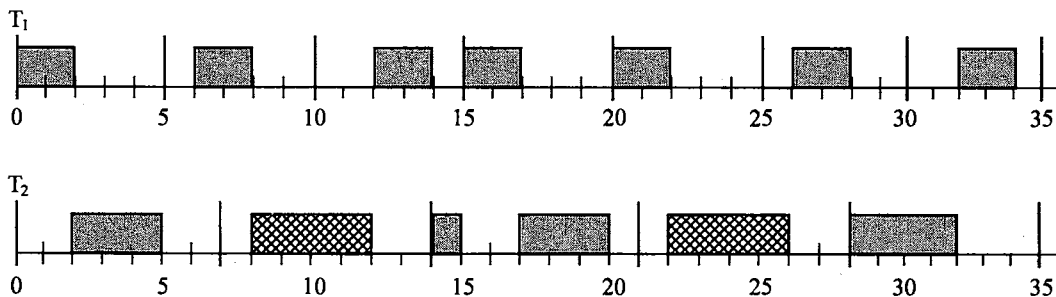


图 3-31 EDF 算法未超时的例子

在  $t_{10}$  时刻，任务  $T_2$  已经运行了两个时间单位，还需要再运行两个时间单位，而任务  $T_1$  的第三个运行周期又开始了。此时， $T_1$  的截止时间是  $t_{15}$ ， $T_2$  的截止时间是  $t_{14}$ ，所以  $T_2$  继续运行。在  $t_{14}$  时刻，任务  $T_2$  开始了它的第三个运行周期，但是在  $t_{15}$  时刻， $T_1$  也要开始它的第四个运行周期。此时， $T_1$  的截止时间是  $t_{20}$ ， $T_2$  的截止时间是  $t_{21}$ 。因此， $T_1$  就打断



了  $T_2$  的运行, 抢占了 CPU。就这样一直进行下去, 每当有一个任务处于就绪状态时, 就重新计算各个任务的优先级, 选择截止时间最近的任务去运行。最后, 这两个任务都能在自己的时间期限内完成, 不会发生超时的现象。

与静态优先级调度算法相比, EDF 算法的显著优点在于它的可调度上限为 100%, 从而使 CPU 的计算能力能够被充分利用起来。EDF 也存在不足之处, 就是在实时系统中不容易实现。而且, 与 RMS 相比, EDF 需要更大的调度开销, 需要在系统运行的过程中动态地计算各个任务的优先级。

### 3.3.6 任务间的同步与互斥

#### 1. 任务之间的关系

在一个嵌入式应用系统中往往包含有多个任务, 它们在系统的硬件平台和操作系统提供的软件平台上运行。这些任务之间主要有以下几种关系。

- 相互独立: 任务之间没有任何的关联关系, 互不干预、互不往来。唯一的相关性就是它们都需要去竞争 CPU 资源。
- 任务互斥: 除了 CPU 之外, 这些任务还需要共享其他的一些硬件和软件资源, 而这些资源由于种种原因, 在某一时刻只允许一个或几个任务去访问。因此当这些任务在访问共享资源的时候可能会相互妨碍。
- 任务同步: 任务之间存在着某种依存关系, 需要协调彼此的运行步调。
- 任务通信: 任务之间存在着协作与分工, 需要相互传递各种数据和信息, 才能完成各自的功能。

在嵌入式操作系统当中, 对于任务间的第一种关系, 主要是靠调度器来进行协调。而对于其他的几种关系, 操作系统必须提供一些机制, 让各个任务能够相互通信、协调各自的行为, 以确保系统能够顺利、和谐地运行。

#### 2. 任务互斥

考虑如下的一个例子: 假设在系统中有两个任务, 它们的程序如图 3-32 所示。如果在这两个任务执行之前, count 变量的值为 1。那么当这些任务完成后, count 变量的值是多少?

任务 1	任务 2
tmp1 = count;	tmp2 = count;
tmp1 ++;	tmp2 = tmp2+2;
count = tmp1;	count = tmp2;

图 3-32 任务互斥的例子

由于这两个任务是并行执行的,所以可能有三种不同的执行序列,如图 3-33 所示。在第一种情况下,任务 1 首先执行了第一条语句,把 tmp1 赋值为 1。然后发生了一个中断,任务 2 开始运行,它的执行结果是把 count 修改为 3。但是当它执行完后,又回到任务 1。它先把 tmp1 加 1,变成 2,再把它赋给 count,所以 count 变量最后的值是 2。

在第二种情况下,任务 2 首先执行了第一条语句,把 tmp2 赋值为 1。然后发生了一个中断,任务 1 开始运行,把 count 修改为 2。最后回到任务 2,又把 count 修改为 3。

在第三种情况下,任务 1 先执行,把 count 修改为 2。然后任务 2 执行,把 count 修改为 4,所以最后的结果是 4。

由此可见,在多任务并行执行的情况下,如果两个任务共享相同的软硬件资源(在本例中是 count 变量),可能会发生意想不到的问题。

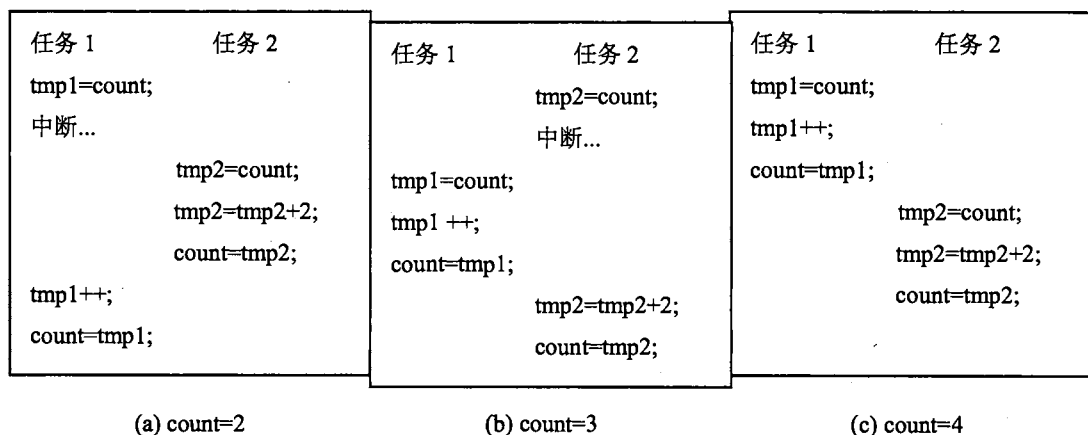


图 3-33 任务互斥的结果

在多道程序操作系统当中,两个或多个任务对同一个共享数据进行读写操作,最后的结果是不可预测的,它取决于各个任务的具体运行情况。我们把这种现象叫做竞争条件(race condition)。那么如何解决竞争条件的问题呢?既然问题产生的根源在于两个或者多个任务对某一个共享数据同时进行读写操作,那么解决的办法很简单,就是在同一个时刻,只允许一个任务来访问这个共享数据。也就是说,如果当前已经有一个任务正在访问这个共享数据,那么其他的任务暂时都不能访问,只能等它先用完。这就是任务之间的互斥。

我们可以用一种抽象的形式来表示这个问题。把一个任务在运行过程中所做的各种事情分为两类:第一类是任务内部的计算或其他的一些事情,这些事情肯定不会导致竞争条件的出现;第二类是对共享资源进行访问,这些访问可能会导致竞争条件的出现。我们把

相应的那一部分程序称为是临界区 (critical section)，把需要互斥访问的共享资源称为是临界资源 (critical resource)。这样，如果能够设计出某种方法，使得任何两个任务都不会同时进入到它们的临界区当中，那么就可以避免竞争条件的出现。不过，这只是一个最基本的要求，在具体实现的时候，还必须考虑其他的一些问题。为此，人们提出了实现互斥访问的四个条件。

- 在任何时候最多只能有一个任务位于它的临界区当中。
- 不能事先假定 CPU 的个数和系统的运行速度。
- 如果某一个任务没有位于它的临界区当中，它不能妨碍其他任务去访问临界资源。
- 任何一个任务进入临界区的请求必须在有限的时间内得到满足，不能无限期等待。

### 3. 任务互斥的解决方案

#### (1) 关闭中断法

为了实现任务之间的互斥，最简单的办法就是把中断关掉。具体来说，当一个任务进入它的临界区之后，首先把中断关闭掉，然后就可以去访问共享资源。当它从临界区退出时，再把中断打开。

关闭中断可以有效地实现任务之间的互斥。对于操作系统而言，它可以认为是由中断来驱动的，只有当发生中断的时候，包括时钟中断、I/O 中断、系统调用等，操作系统才能得到控制权，才能进行任务切换。如果当前任务把中断关闭了，除非它主动让出 CPU，否则将不会发生任务的切换，别的任务将无法运行。在这种情况下，当前任务就可以很方便地去访问共享资源，不用担心别的任务来跟它竞争。

关闭中断法虽然简单有效，但也有它的缺点。首先，这种方法具有一定的风险。当任务把中断关闭后，如果由于种种原因不能及时地打开中断，那么整个系统就可能陷入崩溃的状态。其次，这种方法的效率不高。因为我们的初衷，只是想阻止那些试图访问共享资源的任务，以实现对该资源的互斥访问。但是关闭中断后，所有的任务都被阻止了，不论是竞争对手，还是毫不相关的任务，都被拒绝，无法运行。因此，关闭中断法不能作为一种普遍适用的互斥实现方法，它主要用在操作系统的内核当中，使内核在处理一些关键性的敏感数据时，不会受到其他任务的干扰。

#### (2) 繁忙等待法

实现任务间互斥，也可以采用繁忙等待 (busy waiting) 的策略。其基本思路是：当一个任务想要进入它的临界区时，首先检查一下是否允许它进入，若允许，就直接进入；若不允许，就在那里循环地等待。

在具体实现上，有多种基于繁忙等待的实现方案，如加锁标志位法、强制轮流法、Peterson 算法、TSL 指令等。这些方法可以抽象为如图 3-34 当中的伪代码形式。当一个任务需要进入临界区时，不断地用 while 语句来测试一个标志位，看能否进入，如果不能的

话,就循环等待,直到允许进入。在退出临界区的时候,还要把标志位清除掉。这类方法的共同点就是在测试能否进入临界区的时候,使用的是while 循环语句,不断地执行测试指令,这样就浪费了大量的 CPU 时间。另外,这种方法还有一个问题,它只能处理单一共享资源的情况。如果在系统中,某种类型的共享资源有  $N$  份实例,则在任何时刻,最多应该允许  $N$  个任务同时进入临界区,去访问这种资源。但繁忙等待法无法处理此类问题。

```
while (TestAndSet(lock));  
临界区代码;  
lock = FALSE;  
非临界区代码
```

图 3-34 基于繁忙等待的互斥方法

#### 4. 信号量

信号量是 1965 年由著名的荷兰计算机科学家 Dijkstra 提出的,其基本思路是使用一种新的变量类型,即信号量(semaphore)来记录当前可用资源的数量。

在信号量的具体实现上,有两种不同的方式。

- 方式一:要求信号量的取值必须大于或等于 0。如果信号量的值等于 0,表示当前已没有可用的空闲资源;如果信号量的值大于 0,则该值就代表了当前可用的空闲资源数量;
- 方式二:信号量的取值可正可负。如果是正数或 0,其含义与方式一是相同的;如果是负数,则它的绝对值就代表正在等待进入临界区的任务个数。

信号量是由操作系统来维护的,任务不能直接去修改它的值,只能通过初始化和两个标准原语(即 P、V 原语)来对它进行访问。在初始化时,可以指定一个非负整数,即空闲资源的总数。所谓的原语,通常由若干条语句组成,用来实现某个特定的操作,并通过一段不可分割或不可中断的程序来实现其功能。原语是操作系统内核的一个组成部分,必须在内核态下执行。原语的不可中断性是通过在其执行过程中关闭中断来实现的。

P、V 原语作为操作系统内核代码的一部分,是一种不可分割的原子操作。它们在运行时,不会被时钟中断所打断。另外,在 P、V 原语中包含有任务的阻塞和唤醒机制,因此,当任务在等待进入临界区的时候,会被阻塞起来,而不会去浪费 CPU 时间。

P 原语中的字母 P,是荷兰语单词“测试”的首字母。它的主要功能是申请一个空闲的资源,把信号量的值减 1。如果成功的话,那么就退出原语;如果失败的话,那么这个任务就会被阻塞起来。V 原语当中的字母 V,是荷兰语单词“增加”的首字母。它的主要功能是释放一个被占用的资源,把信号量的值加 1,如果发现有被阻塞的任务,就从中选择一个把它唤醒。

在操作系统当中,如何实现信号量和 P、V 原语操作呢?图 3-35 和图 3-36 给出了一种具体的实现方案。图 3-35 所示是信号量结构体类型的定义,一个信号量由两部分内容组



成：一个是整型的计数变量 `count`，如果它的值大于 0，就表示当前空闲资源的个数；如果它的值小于 0，它的绝对值就表示位于阻塞队列当中的任务个数。第二部分内容是一个阻塞队列 `queue`，表示由于等待这个信号量而被阻塞起来的各个任务，它们被指针一个个地串起来，形成了一个队列。队列当中的每一个节点表示相应任务的任务控制块 `TCB`。有了这个数据类型的定义后，就可以用它去声明所需要的信号量了。

```
typedef struct
{
    int count;
    struct TCB *queue;
} semaphore;
```

图 3-35 信号量的结构体类型

图 3-36(a)所示是 P 原语的一个实现，它其实就是一个函数，形参是一个信号量 `S`。首先把 `S` 的计数变量减 1，即申请一个空闲资源。然后判断，如果这个计数变量的值小于 0，这说明在本函数被调用之前，就已经没有空闲资源了。因此就把调用这个原语的任务，把它的状态从运行状态变为阻塞状态，并加入到信号量 `S` 的阻塞队列的末尾。接下来，通知调度器重新选择一个就绪任务去运行。

图 3-36(b)所示是 V 原语的一个实现，它的形参也是信号量 `S`。首先把 `S` 的计数变量加 1，即释放一个被占用的资源。然后判断，如果这个计数变量的值小于或等于 0，就说明在本函数被调用之前，在这个信号量的阻塞队列当中，存在着被阻塞的任务。因此就从 `S` 的阻塞队列当中，取出一个任务，把它的状态从阻塞状态改为就绪状态，并插入到系统的就绪队列中去。

P(semaphore S)

```
{
    --S.count; //申请一个资源
    if(S.count < 0) //没有空闲资源
    {
        将当前任务阻塞起来，加入到阻塞
        队列 S.queue 末尾;
        调度新任务运行;
    }
}
```

(a) P 原语

V(semaphore S)

```
{
    ++S.count; //释放一个资源
    if(S.count <= 0) //有任务被阻塞
    {
        从阻塞队列 S.queue 中取出一个任务;
        将该任务改为就绪状态，插入就绪队列
    }
}
```

(b) V 原语

图 3-36 P、V 原语的实现

利用操作系统提供的信号量机制，可以方便、有效地实现对临界资源的互斥访问。如图 3-37 所示，首先定义了一个信号量类型的变量  $S$ ，并对它进行初始化。如果共享资源的个数为 1，就把  $S$  初始化为 1。这样，在任何时刻，只允许一个任务进入临界区。如果共享资源的个数为  $N$ ，就把  $S$  初始化为  $N$ 。这样，在同一时刻，可以允许  $N$  个任务同时进入临界区。然后在程序当中，当需要访问临界资源的时候，首先用一个  $P$  原语，判断现在能否进入临界区。如果能的话， $P$  原语将顺利结束，因而当前任务就能继续往下运行，并进入临界区。如果现在已经没有空闲的临界资源，那么当前任务就会在  $P$  原语的内部被阻塞起来，加入到  $S$  信号量的阻塞队列当中去。一直等到另外的某个任务退出了临界区，它就会调用  $V$  原语，从阻塞队列中，把任务唤醒。当任务被唤醒后，就从  $P$  原语的后面继续往下执行，从而进入了临界区。另外，当一个任务在退出临界区时，一定要使用  $V$  原语来释放资源，并唤醒正在等待该资源的其他任务。

```
semaphore S;  
S.count = N; //N 表示资源个数  
P(S);  
临界区;  
V(S);  
非临界区;
```

图 3-37 基于信号量的任务互斥

采用信号量来实现任务之间的互斥，优点有两个：一是可以设置信号量的计数值，从而允许多个任务同时进入临界区；二是当一个任务暂时无法进入临界区时，它会被阻塞起来，从而让出 CPU 给其他的任务。

大多数嵌入式操作系统都提供了信号量的机制，用户可以通过函数调用的方式去使用。例如，在 VxWorks 操作系统当中，系统提供了三种类型的信号量：二值信号量、互斥信号量和计数型信号量。每种类型的信号量都有相应的创建函数。此外，系统还提供了两个信号量操作函数： $\text{semTake}$  函数，类似于  $P$  操作； $\text{semGive}$  函数，类似于  $V$  操作。在另外一个嵌入式操作系统  $\mu\text{C}/\text{OS-II}$  当中， $\text{osSemCreate}$  函数表示创建一个信号量， $\text{osQuePend}$  函数类似于  $P$  操作， $\text{osSemPost}$  类似于  $V$  操作。

## 5. 任务同步

一般来说，一个任务相对于另一个任务的运行速度是不确定的，也就是说，任务是在异步环境下运行的。每个任务都以各自独立的、不可预知的速度向前推进。但是在有些时候，在两个或多个任务中执行的某些代码片断之间，可能存在着某种时序关系或先后关系，所以这些任务必须协同合作、相互配合，使各个任务按一定的速度运行，以共同完成某一项工作。这就是任务间的同步。

要实现任务之间的同步，可以使用信号量机制，通过引入  $P$ 、 $V$  操作来设定两个任务在运行时的先后顺序。例如，可以把信号量视为某个共享资源的当前个数，然后由一个任

务负责生成这种资源，而另一个任务则负责消费这种资源，这样，就构成了这两个任务之间的先后顺序。在具体实现上，一般把信号量的初始值设为  $N$ ， $N$  大于或等于 0。然后在一个任务的内部使用  $V$  原语，增加资源的个数，而在另一个任务的内部使用  $P$  原语，减少资源的个数，从而实现这两个任务之间的同步关系。

例如，假设有两个任务  $T_1$  和  $T_2$ 。 $T_1$  做的事情主要是代码片断 A， $T_2$  做的事情主要是代码片断 B。任务  $T_1$  和  $T_2$  同时位于系统当中，相互独立地运行。但由于这两个任务之间存在某种同步关系，要求代码 A 必须先执行，然后代码 B 才能执行。比如说，A 负责采集信号，B 负责对这些信号进行处理，显然，只有当 A 把信号采集进来之后，B 才能去处理。由于在多道程序的操作系统当中，各个任务的执行是相互独立的，系统可能先调度任务  $T_1$  去运行，也可能先调度任务  $T_2$  去运行，这是由调度算法来决定的。因此，我们无法保证任务  $T_1$  肯定比任务  $T_2$  先执行。但是为了实现任务之间的同步，我们必须保证，无论是任务  $T_1$  先执行还是任务  $T_2$  先执行，从最后的结果来看，代码片断 A 必须先执行，然后代码片断 B 才能执行。

如何用信号量来实现这种同步呢？如图 3-38 所示，首先定义了一个信号量  $S$ ，并把它计数值初始化为 0。然后在任务  $T_1$  的代码片断 A 的后面，加了一个  $V$  操作，在任务  $T_2$  的代码片断 B 的前面，加了一个  $P$  操作，这样就能够实现 A 和 B 之间的同步关系。具体来说：

- 假设任务  $T_1$  先被调度执行。它先执行 A，然后执行  $V$  操作，把信号量  $S$  的值变成 1，然后就结束了。接下来，任务  $T_2$  被调度执行，它先执行  $P$  操作，由于此时信号量的值为 1，所以顺利通过。然后再执行 B。因此最后的运行结果就是先 A 后 B。
- 假设任务  $T_2$  先被调度执行。由于信号量的初值为 0，所以当它在调用  $P$  操作时，会被阻塞起来，加入到信号量  $S$  的阻塞队列，并交出了 CPU。然后任务  $T_1$  执行，它先执行 A，然后用  $V$  操作释放信号量，唤醒任务  $T_2$ 。这样，当  $T_2$  重新开始运行的时候，就直接去执行 B。因此最后的结果还是先 A 后 B。

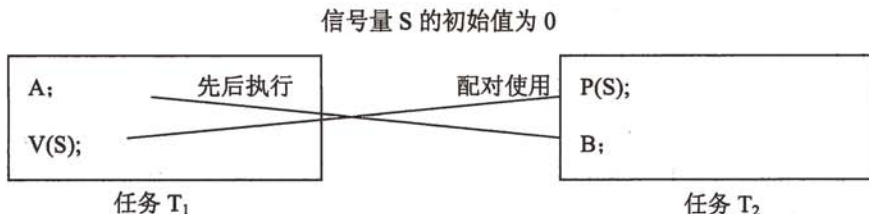


图 3-38 任务间同步的例子

图 3-39 所示是任务间同步的另一个例子。假设信号量  $S$  的初始值为 1，那么当任务  $T_1$  和  $T_2$  在运行的时候，可能出现的执行序列有 AB、ABAB、AAAABBBB、BABA 等，不

可能出现的执行序列有 BBA、ABBB 等。实际上，本例的同步关系为：在任何时刻，B 的个数最多只能比 A 的个数多 1。

信号量 S 的初始值为 1

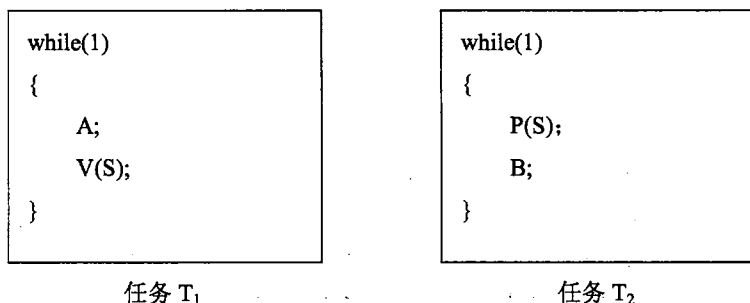


图 3-39 任务间同步的另一个例子

## 6. 死锁

在一组任务当中，由于每个任务都占用着若干个资源，同时又在等待其他任务所占用的资源，从而造成所有任务都无法进展下去的现象，称这种现象称为死锁（deadlock），这一组相关的任务称为死锁任务。在死锁状态下，每个任务都动弹不得，既无法运行，也无法去释放所占用的资源，它们互为因果、相互等待。

死锁的产生有四个必要条件，只有当这四个条件同时成立时，才会出现死锁。

- 互斥条件：在任何时刻，每一个资源最多只能被一个任务所使用。
- 请求和保持条件：任务在占用若干个资源的同时又可以请求新的资源。
- 不可抢占条件：任务已经占用的资源不会被强制性拿走，而必须由该任务主动释放。
- 环路等待条件：存在一条由两个或多个任务所组成的环路链，其中每一个任务都在等待环路链中下一个任务所占用的资源。

除了资源的竞争之外，PV 操作使用不当也会引起死锁，图 3-40 所示是一个例子。在系统中，定义了两个信号量 S 和 Q，它们的初始值都是 1。两个任务 T<sub>1</sub> 和 T<sub>2</sub>，假设 T<sub>1</sub> 先被调度执行，它顺利地通过了 P(S)操作，并使 S 的值变为 0。假设这时发生了一次时钟中断，任务 T<sub>2</sub> 被调度执行。它顺利地通过了 P(Q)操作，并将 Q 的值变为 0。接着在执行 P(S)操作时，由于 S 的值已经是 0，因此 T<sub>2</sub> 在这里被阻塞起来，并让出 CPU。然后任务 T<sub>1</sub> 重新开始运行，但是当它执行到 P(Q)时，由于 Q 的值已经为 0，所以 T<sub>1</sub> 也被阻塞起来。这样一来，任务 T<sub>1</sub> 和 T<sub>2</sub> 都处于阻塞状态，都在等待对方释放信号量，就成为了一种死锁的状态。

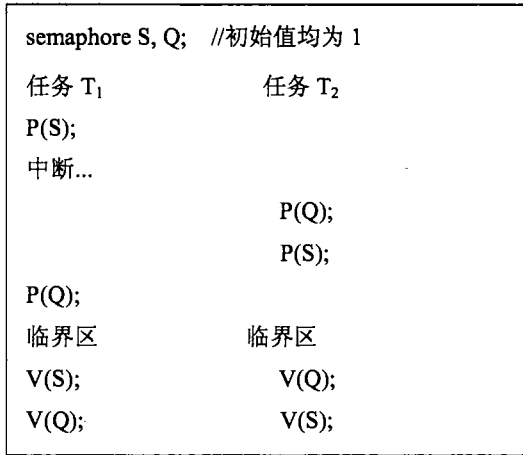


图 3-40 PV 操作引发的死锁

## 7. 信号

任务间同步的另一种方式是异步信号 (asynchronous signal)。在两个任务之间，可以通过相互发送信号的方式，来协调它们之间的运行步调。

所谓的信号，指的是系统给任务的一个指示，表明某个异步事件已经发生了。该事件可能来自于外部（如其他的任务、硬件或定时器），也可能来自于内部（如执行指令出错）。异步信号管理允许任务定义一个异步信号服务例程 ASR。与中断服务程序不同的是，这里的 ASR 是与特定的任务相对应的。当一个任务正在运行的时候，如果它收到了一个信号，将暂停执行当前的指令，转而切换到相应的信号服务例程去运行。不过这种切换不是任务之间的切换，因为信号服务例程通常还是在当前任务的上下文环境中运行的。另外，信号的发送与任务的状态无关，当一个任务收到一个信号时，它可能处于运行状态，也可能处于就绪状态，而信号的接收并不会对任务的当前状态有任何影响。如果任务在收到信号时并未处于运行状态，那么当它下次被调度运行时，就会去调用相应的 ASR 来处理此次信号。因此，信号机制也可以称为软中断机制。

信号机制与中断处理机制非常相似，但又各有不同。它们的相同点如下。

- 都具有中断性：在处理中断和异步信号时，都要暂时地中断当前任务的运行。
- 都有相应的服务程序。
- 都可以屏蔽响应：外部硬件中断可以通过相应的寄存器操作来屏蔽，任务也能够选择不对异步信号进行响应。

信号机制与中断机制的不同点如下。

- 中断是由硬件或特定的指令产生的，而信号是由系统调用产生的。

- 中断触发后，硬件会根据中断向量找到相应的处理程序去执行；而信号则通过发送信号的系统调用来触发，但系统不一定马上对它进行处理。
- 中断处理程序在系统内核的上下文中运行，是全局的；而信号处理程序在相关任务的上下文中运行，是任务的一个组成部分。

### 3.3.7 任务间通信

任务间通信（intertask communication）指的是任务之间为了协调工作，需要相互交换数据和控制信息。任务之间的通信可以分为两种类型。

- 低级通信：只能传递状态和整数值等控制信息。例如，用来实现任务间同步与互斥的信号量机制和信号机制都是一种低级通信方式。这种方式的优点是速度快，缺点是传送的信息量非常少，如果要传递较多信息，就得进行多次通信。
- 高级通信：能够传送任意数量的数据，主要包括共享内存、消息传递和管道三类。

#### 1. 共享内存

共享内存（shared memory）指的是各个任务共享它们地址空间当中的某些部分，在此区域，可以任意读写和使用任意的数据结构，把它看成是一个通用的缓冲区。一组任务向共享内存中写入数据，另一组任务从中读出数据，通过这种方式来实现它们之间的信息交换。

在有些嵌入式操作系统中，不区分系统空间和用户空间，整个系统只有一个地址空间，即物理内存空间，系统程序和各个任务都能直接对所有的内存单元进行随意访问。在这种方式下，内存数据的共享就变得更加容易了，如图 3-41 所示。

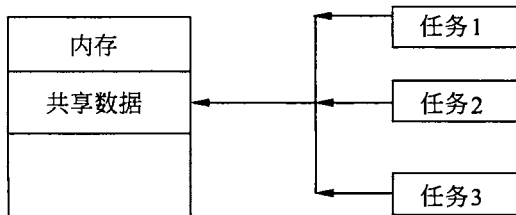


图 3-41 多个任务共享内存空间

在使用共享内存来传送数据的时候，通常要与某种任务间的互斥机制结合起来，以免发生竞争条件的现象，确保数据传送的顺利进行。

#### 2. 消息传递

消息（message）是内存空间中一段长度可变的缓冲区，其长度和内容均由用户定义。从操作系统的角度来看，所有的消息都是单纯的字节流，既没有确切的格式，也没有特定的含义。对消息内容的解释是由应用来完成的，应用根据自定义的消息格式，将消息解释成特定的含义，如某种类型的数据、数据块的指针或空。

消息传递 (message passing) 指的是任务与任务之间通过发送和接收消息来交换信息。

消息机制由操作系统来维护, 包括定义寻址方式、认证协议、消息的数量等。一般提供两个基本的操作: **send** 操作, 用来发送一条消息; **receive** 操作, 用来接收一条消息。如果两个任务想要利用消息机制来进行通信, 它们首先要在两者之间建立一个通信链路, 然后就可以使用 **send** 和 **receive** 操作来发送和接受消息了。

任务之间的通信方式可以分为直接通信和间接通信两种。

### (1) 直接通信

在直接通信方式下, 通信双方必须明确知道与之通信的对象。这种方式采用类似下面的通信原语。

- **send (P, message)**: 发送一条消息给任务 P。
- **receive (Q, message)**: 从任务 Q 那里接收一条消息。如果没有收到消息, 可以阻塞起来等待消息的到来, 也可以立即返回。

在通信双方之间存在一条通信链路, 该链路具有如下特征。

- 通信链路是自动建立的, 由操作系统来维护。
- 每条链路只涉及一对相互通信的任务, 每对任务之间仅存在一条链路。
- 通信链路可以是单向或双向的。

### (2) 间接通信

在间接通信方式下, 通信双方不需要指出消息的来源或去向, 而是通过共享的邮箱 (mailbox) 来发送和接收消息, 每个邮箱都有一个唯一的标识。这种方式采用类似下面的通信原语。

- **send (A, message)**: 发送一条消息给邮箱 A。
- **receive (A, message)**: 从邮箱 A 接收一条消息。

间接通信的特点:

- 对于一对任务, 只有当它们共享一个公共邮箱时才能进行通信。
- 一个邮箱可以被多个任务访问, 每对任务也可以使用多个邮箱来通信。
- 通信可以是单向或双向的。

邮箱只能存放单条消息, 它提供了一种低开销的消息传递机制, 其状态只有两种: 空或满。另外一种间接通信机制是消息队列 (message queue), 它与邮箱是类似的, 但可以同时存放若干条消息, 提供了一种任务间缓冲通信的方法。如图 3-42 所示, 发送消息的任务将消息放入队列, 而接收消息的任务则将消息从队列中取出。

## 3. 管道

管道 (pipe) 通信由 UNIX 首创, 由于其有效性, 后来的一些系统相继引入了管道技术。管道通信以文件系统为基础, 所谓管道即连接两个任务之间的一个打开的共享文件, 专用于任务之间的数据通信。发送任务从管道的一端写入数据流, 接收任务从管道的另一

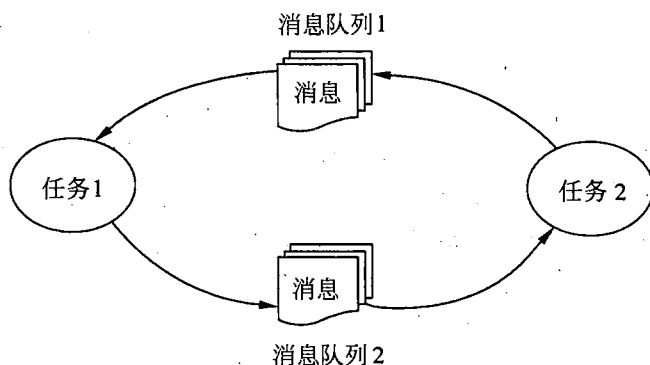


图 3-42 消息队列示意图

端按先进先出的顺序读出数据流。管道的读写操作即为普通的文件读写操作，数据流的长度和格式没有限制。

## 3.4 存储管理

### 3.4.1 存储管理概述

不同的嵌入式系统采用不同的存储管理方式，有的简单，有的复杂，这与实际的应用领域及硬件环境密切相关。在强实时应用领域，存储管理方法就比较简单，甚至不提供存储管理功能。而一些实时性要求不高，可靠性要求比较高且系统比较复杂的应用在存储管理上就相对复杂一些，可能需要实现对操作系统或任务的保护。

通常在设计存储管理的时候，需要考虑如下一些因素。

- 硬件条件：例如是否有存储管理单元 MMU。
- 实时性要求：是硬实时、软实时还是分时系统。
- 系统规模大小、复杂程度、性能要求。
- 可靠性要求：是否需要内存保护。

#### 1. 内存保护

在嵌入式微处理器当中，存储管理单元（Memory Management Unit, MMU）提供了一种内存保护的硬件机制。操作系统通常利用 MMU 来实现系统内核与应用程序的隔离，以及应用程序与应用程序之间的隔离。这样可以防止应用程序去破坏操作系统和其他应用程序的代码和数据，防止应用程序对硬件的直接访问。内存保护包含两个方面的内容：一是防止地址越界，每个应用程序都有自己独立的地址空间，当一个应用程序要访问某个内存单元时，由硬件检查该地址是否在限定的地址空间内，如果不是的话就要进行地址越界处



理；二是防止操作越权，对于允许多个应用程序共享的某块存储区域，每个应用程序都有自己的访问权限，如果违反了权限规定，则要进行操作越权处理。

在一些低端的、比较简单的嵌入式应用中，出于成本的考虑，CPU 的运算速度比较慢，不具备 MMU 的功能，而且存储空间有限，系统软件的代码量受到严格限制，例如，在有的嵌入式系统中，操作系统只占据几 KB 的空间。另外，系统的安全性、可靠性的要求也不是很高，即使系统崩溃也不会造成重大的损失。对于这种类型的应用，它们对内存保护方面的要求非常低，通常不需要内核提供内存保护机制。

在一些高端的、比较复杂的嵌入式应用中，允许投入更多的硬件和软件成本——CPU 的运行速度非常快，并且具备 MMU 功能。对于这种类型的应用，它们对安全性和可靠性的要求比较高，通常要求内核提供内存保护机制。

## 2. 实时性要求

系统的实时性要求也会影响到存储管理的实现方式。在实现一个嵌入式实时内核时，为了确保系统的实时性，在内存管理方面需要考虑如下的因素。

- 速度快：存储管理方面的开销不能太大，尤其是在一些低配置的硬件平台上，不能去使用一些比较复杂的存储管理方案。
- 确定性：对于每一项工作都要有明确的实时约束，即必须在某个限定的时刻之前完成。因此，在实时系统中，一般不采用虚拟存储管理技术。原因在于：在虚拟存储管理中，可能会发生缺页中断，这就需把保存在外围存储介质中的页面调入内存，而这部分工作所需要的时间难以预测，因而不利于系统的确定性。

## 3.4.2 实模式与保护模式

在嵌入式操作系统当中，常见的存储管理方案可以分为两大类：实模式方案和保护模式方案。

### 1. 实模式方案

实模式方案也叫内存的平面使用模式。在这种存储管理方式下，系统将关闭 MMU 或者根本就没有 MMU。

实模式方案的主要特点是：

- 不划分“系统空间”和“用户空间”，整个系统只有一个地址空间，即物理内存地址空间，应用程序和系统程序都能直接对所有的内存单元进行随意访问，无须进行地址映射。
- 操作系统的内核与外围应用程序之间不再有物理的边界，在编译连接后，两者通常被集成在同一个系统文件中。
- 系统中所说的“任务”或“进程”，实际上全都是内核线程。对于这些线程来说，

只有运行上下文和栈是独享的，其他资源都是共享的。

实模式方案的优点是：简单、性能好，而且存储管理的开销比较确定，这对于实时系统来说是比较重要的。它的缺点是：没有存储保护、安全性差，在应用程序中出现的任何一个小错误或蓄意攻击都有可能整个系统的崩溃。因此，它比较适合于规模较小、简单和实时性要求较高的系统。事实上，大多数传统的嵌入式操作系统均采用此模式。

以 Samsung S3C44B0X 微处理器为例，这是三星公司专为手持设备和一般应用提供的高性价比和高性能的微控制器解决方案，它使用 ARM7TDMI 核，工作在 66MHz 频率下。

在 S3C44B0X 的地址空间中，有八个存储体，每个存储体可达 32MB，总共可达 256MB。在这八个存储体中，Bank0~Bank5 可支持 ROM 和 SRAM，Bank6、Bank7 可支持 ROM、SRAM 和 FP/EDO/SDRAM 等。

在一个典型的系统中，各个存储体的分配情况如下。

- Bank0: BIOS, 512KB×2 Flash。
- Bank1: 16MB Flash 硬盘。
- Bank2: USB 接口。
- Bank3: LCD 显示模块。
- Bank6: 8MB 系统内存 SDRAM。
- Bank4、Bank5、Bank7: 保留。

图 3-43 所示是系统复位后 S3C44B0X 的存储器映射表。

在系统复位时，程序计数器（PC 寄存器）被设置成 0，使系统跳转到 0x00000000 处开始运行。此地址对应的是 Bank0，即 1MB 的线性 Flash，里面存放的是系统的初始化程序，它负责配置处理器的结构、工作模式以及自动检测嵌入式控制器的各个硬件是否工作正常。然后，它把存储在 16MB Flash 里面的 system.bin 文件复制到 0x0C000000 地址中，此地址是系统的 8MB 内存的首地址。然后引导程序把 PC 寄存器指向 0x0C000000 地址，系统开始运行。system.bin 是嵌入式操作系统引导的执行文件，编译以后的操作系统和应用程序都被集成在这个文件当中。

如图 3-44 所示，在实模式存储管理方案下，嵌入式系统的内存地址空间的布局一般可以分为五个段。

- .text: 代码段。包含了操作系统和应用程序的所有代码。
- .data: 数据段。存放了操作系统和应用程序当中所有带有初始值的全局变量。
- .bss: bss 段。存放了操作系统和应用程序当中所有未带初始值的全局变量。
- 堆空间: 动态分配的内存空间。在系统运行时，可以通过类似于 malloc/free 之类的函数来申请或释放一段连续的内存空间。
- 栈空间: 保存运行上下文以及函数调用时的局部变量和行参。

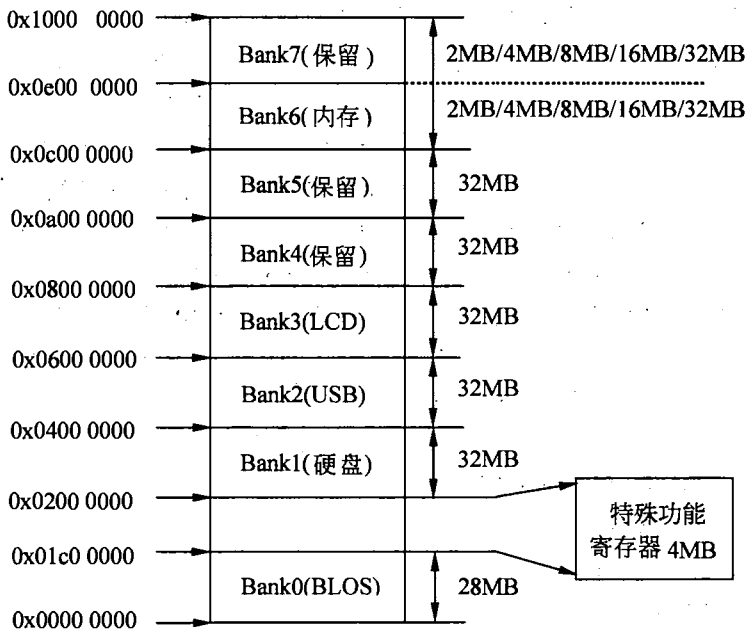


图 3-43 复位后 S3C44B0X 的存储器映射表

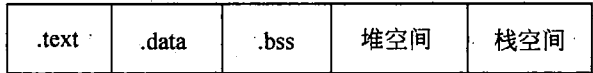


图 3-44 内存布局

图 3-45 所示通过一个例子演示了各个段的含义。在该程序中，全局变量 `gvCh` 和 `gvShort` 由于没有设置初始值，所以放在 `.bss` 段当中；全局变量 `gvInt` 和 `gvLong` 有初始值，所以放在 `.data` 段当中；而 `main` 函数经过编译以后得到的机器代码，则存放在 `.text` 代码段当中；指针 `p` 和数组 `array` 都是 `main` 函数的局部变量，所以存放在栈当中；而 `malloc` 函数分配的存储空间，则位于堆空间当中。

对于 `.text`、`.data` 和 `.bss` 段，它们的大小在编译时即可确定，所以称为静态段。而对于堆和栈，它们的大小在编译时不能确定，而且会随着系统的运行而发生变化，所以称为动态段。不过，在有些嵌入式系统中，任务的栈空间是以静态数组的方式来实现的，存放在 `.bss` 段当中，其大小是固定的。

显然，在实模式存储管理方案下，主要的工作在于堆空间的管理，即如何来管理空闲的堆空间、如何来分配内存、如何来回收内存等，这通常可以采用 3.4.3 小节当中的分区存储管理方案来实现。

```
#include <malloc.h>
unsigned char gvCh;
unsigned short gvShort;
unsigned int gvInt = 0x12345678;
unsigned long gvLong = 0x87654321;
void main(void)
{
    unsigned char array[10], *p;
    p = malloc(10*sizeof(char));
    while (1);
}
```

图 3-45 各个段的内容举例

## 2. 保护模式方案

保护模式方案指的是在处理器中必须要有 MMU 硬件并启用。它的主要特点如下。

- 系统内核和用户程序都有各自独立的地址空间：操作系统和 MMU 共同合作，完成逻辑地址到物理地址的映射。
- 具有存储保护功能：每个应用程序只能访问自己的地址空间，不能去破坏操作系统和其他应用程序的代码和数据。对于共享的内存区域，也必须按照规定的权限规则来访问。

保护模式方案的优点是安全性和可靠性较好，它比较适合于规模较大、较复杂和实时性要求不太高的系统。

### 3.4.3 分区存储管理

在多道程序操作系统当中，同时有多个任务在系统中运行，每个任务都有各自的地址空间。为了实现这种多道程序系统，在存储管理上，最简单的做法就是采用分区存储管理。它的基本思路是：把整个内存划分为两大区域，即系统区和用户区，然后再把用户区划分为若干个分区，分区的大小可以相等，也可以不等，每个任务占用其中的一个分区。这样，就可以在内存当中同时保留多个任务，让它们共享整个用户区，从而实现多个任务的并发运行。在具体实现上，分区存储管理又可以分为两类：固定分区和可变分区。

### 1. 固定分区存储管理

固定分区存储管理的基本思路是：各个用户分区的个数、位置和大小一旦确定后，就固定不变，不能再修改了。例如，在系统启动时，由管理员来手工划分出若干个分区，并确定各个分区的起始地址和大小等参数。然后，在系统的整个运行期间，这些参数就固定下来了，不再改变。另外，为了满足不同程序的存储需要，各个分区的大小可以是相等的，也可以是不相等的。

当一个新任务到来时，需要根据它的大小，把它放置到相应的输入队列中去，等待合适的空闲分区。在具体的实现上，主要有两种实现方式，即多个输入队列和单个输入队列。

图 3-46 (a) 所示是多个输入队列的一个例子。整个内存被分成五个区，包括四个用户分区和一个系统分区。操作系统放在内存地址低端，占用了 100KB。分区 1、分区 2 和分区 3 的大小分别是 100KB、200KB 和 300KB，分区 4 的大小是 100KB。在多个输入队列的方式下，对于每一个用户分区，都有一个相应的输入队列。在分区 1 的输入队列中有三个任务在等待，在分区 2 的输入队列中有一个任务在等待。分区 3 的输入队列是空的，而分区 4 的输入队列中有两个任务在等待。当一个新任务到来时，就把它加入到某一个输入队列中去。要求这个输入队列所对应的分区，是能够装得下该任务的最小分区。例如，假设现在又来了一个新任务，它的大小是 180KB，那么应该把它加入到分区 2 的输入队列中去，因为在当前情形下，能够装得下该任务的只有分区 2 和分区 3，而分区 2 比分区 3 要小，所以它更合适。

这种为每一个分区都设置一个输入队列的做法，有一个很大的缺点，它可能会出现内存利用率不高的问题，即小分区的输入队列是满的，而大分区的输入队列却是空的。例如，在图 3-46 (a) 所示的这个例子中，在分区 1 的输入队列中，有三个任务在等待，而分区 3 的输入队列却是空的。也就是说，一方面，有很多个小的任务在等着进入内存，而另一方面，在内存中却存在着大量的空闲空间。事实上，如果把这 300KB 的空闲空间平均分给这三个任务，那么它们就都能进入内存了。

为了解决这个问题，人们又提出了单个输入队列的方法。它的基本思路是：对于所有的用户分区，只设置一个统一的输入队列，当一个新任务到来时，就把它加入到这个输入队列当中，然后当某个分区变得空闲的时候，就从这个队列中选择合适的任务去占用该分区。在任务的选择上，可以有两种方法：第一种方法是选择离队首最近的、能够装入这个分区的任务，但是这样的话，如果选中的是一个比较小的任务，那么就会浪费大量的内存空间；第二种方法是先搜索整个队列，从中选择能够装入这个分区的最大任务，这样就能尽可能地减少所浪费的空间。

对于固定分区的存储管理方法，它的优点是易于实现，系统的开销比较小；无论是空闲空间的管理，还是内存的分配和回收算法，都非常简单；无论是算法的时间复杂度，还

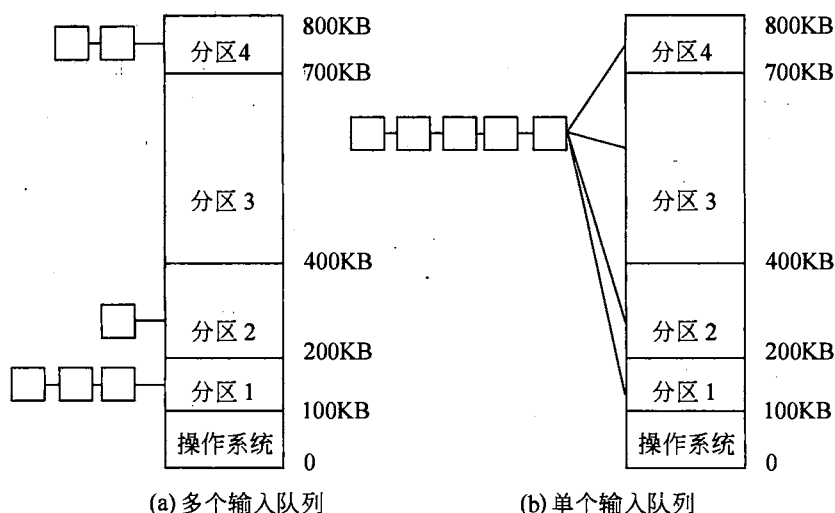


图 3-46 固定分区的输入队列

是空间复杂度，都比较低。因此，系统的管理开销比较小。但是它也有两个主要的缺点：第一，内存的利用率不高，内碎片会造成很大的浪费，所谓的内碎片，就是在任务所占用的分区内部未被利用的空间；第二，分区的总数是固定的，这就限制了并行执行的程序个数，如果一开始只分了  $N$  个分区，那么最多只能有  $N$  个任务在同时运行，这就显得不够灵活。

## 2. 可变分区存储管理

可变分区的基本思路是：分区不是预先划分好的固定区域，而是动态创建的。在装入一个程序时，系统将根据它的需求和内存空间的使用情况来决定是否分配。具体来说，在系统生成后，操作系统会占用内存的一部分空间，这一般放在内存地址的最低端，其余的空间则成为一个完整的大空闲区。当一个程序要求装入内存运行时，系统就会从这个空闲区当中划出一块来，分配给它，当程序运行结束后会释放所占用的存储区域。随着一系列的内存分配和回收，原来的一整块的大空闲区就会形成若干个占用区和空闲区相间的布局。

图 3-47 所示是一个可变分区的例子。在系统当中，整个内存区域有 1024KB。如图 3-47 (a) 所示，在初始化的时候，操作系统占用了内存地址最低端的 128KB，剩下的空间就成为一个完整的大空闲区，总共是 896KB。然后，任务 1 进入了内存，它所需要的内存空间是 320KB，所以紧挨着操作系统，给它分配了一块大小为 320KB 的内存，也就是说，创建了一个新的用户分区，该分区的起始地址是 128KB，大小是 320KB。此时，剩余的内存



空间还有 576KB，而且还是连续的一整块。接下来，任务 2 和任务 3 先后进入内存，它们所需要的内存空间分别是 224KB 和 288KB，所以紧挨着任务 1，给它们分配了两块内存空间，大小分别是 224KB 和 288KB，也就是说，又创建了两个新的用户分区。此时，在内存中总共有 3 个任务，连同操作系统，总共占用了 960KB 的内存空间。因此，在地址空间的最高端，只剩下一小块 64KB 的空闲区域，如图 3-47 (b) 所示。接下来，任务 2 运行结束，系统回收了它所占用的内存分区。然后任务 4 进入内存，它的大小为 128KB，所以就对刚才存放任务 2 的那个分区，一分为二，一部分用来存放任务 4，另一部分是 96KB 的空闲区。接下来，任务 1 也运行结束，系统回收了它所占用的分区。因此，内存空间的最后状态如图 3-47 (c) 所示。从图中可以看出，随着系统地不断运行，空闲区就变得越来越大，越来越碎了，它们被分隔在内存的不同位置，形成了占用区和空闲区交错在一起的局面。

可变分区存储管理的优点是：与固定分区相比，在可变分区当中，分区的个数、位置和大小都是随着任务的进出而动态变化的，非常灵活，当一个任务要进入内存的时候，就在空闲的地方创建一个分区，把它装进来；当任务运行结束后，就把它所占用的内存分区给释放掉。这样，就避免了在固定分区当中由于分区的大小不当所造成的内碎片，从而提高了内存的利用效率。在可变分区的存储管理当中，不会出现内碎片的现象，因为每个分区都是按需分配的，分区的大小正好等于任务的大小。

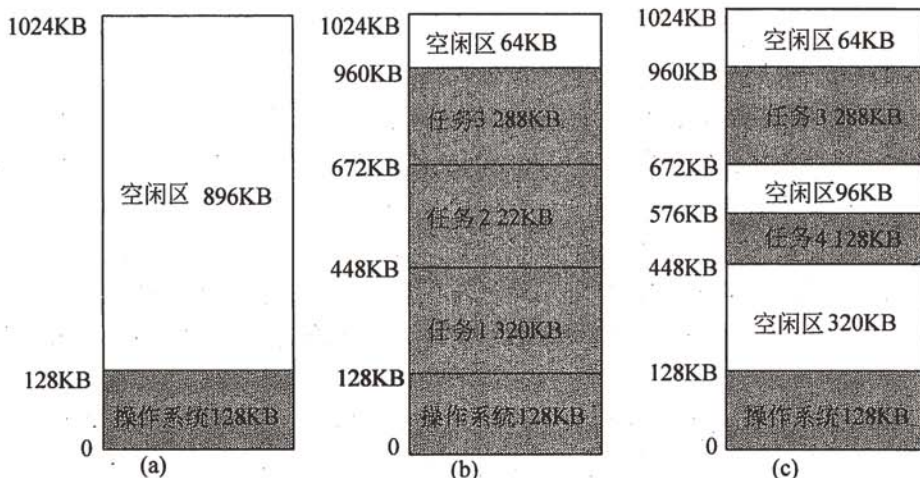


图 3-47 可变分区的例子

可变分区存储管理的缺点是：可能会存在外碎片。所谓的外碎片，就是在各个占用的

分区之间,难以利用的一些空闲分区。这通常是一些比较小的空闲分区。例如,在图 3-47 (c) 当中,对于 64KB 这个空闲区,它只能分配给那些不超过 64KB 的任务,如果所有的任务都大于 64KB 的话,那么它就用不上了,变成了一个外碎片。另外,这种可变分区的办法,使得内存的分配、回收和管理变得更加复杂了。

在具体实现可变分区存储管理技术的时候,需要考虑三个方面的问题:内存管理的数据结构、内存的分配算法以及内存的回收算法。

在内存管理的数据结构上,系统会维护一个分区链表,来跟踪记录每一个内存分区的情况,包括该分区的状态(已分配或空闲)、起始地址、长度等信息。具体来说,对于内存当中的每一个分区,分别建立一个链表节点,记录它的各种管理信息。然后,将这些节点按照地址的递增顺序进行排列,从低到高,形成一个分区链表。

在内存的分配算法上,当一个新任务来到时,需要为它寻找一个空闲分区,其大小必须大于或等于该任务的要求。若是大于要求,则将该分区分割成两个小分区,其中一个分区为要求的大小并标记为“占用”,另一个分区为余下部分并标记为“空闲”。选择分区的先后次序一般是从内存低端到高端。通常的分区分配算法有:最先匹配法、下次匹配法、最佳匹配法和最坏匹配法。

- 最先匹配法 (first-fit): 假设新任务对内存大小的要求为  $M$ , 那么从分区链表的首节点开始, 将每一个“空闲”节点的长度与  $M$  进行比较, 看是否大于或等于它, 直到找到第一个符合要求的节点。然后把它所对应的空闲分区分割为二个小分区: 一个用于装入该任务, 另一个仍然空闲。与之相对应, 把这个链表节点也一分为二, 并修改相应内容。
- 下次匹配法 (next-fit): 这与最先匹配法的思路是相似的, 只不过每一次当它找到一个合适的节点(分区)时, 就把当前的位置记录下来, 然后等下一次新任务到来的时候, 就从这个位置开始继续往下找(到链表结尾时再回到开头), 直到找到符合要求的第一个分区。而不是像最先匹配法那样, 每次都是从链表的首节点开始找。
- 最佳匹配法 (best-fit): 将申请内存的任务装入到与其大小最接近的空闲分区当中。这种算法的最大缺点是分割后剩余的空闲分区将会很小, 直至无法使用, 从而造成浪费。
- 最坏匹配法 (worst-fit): 每次分配时, 总是将最大的空闲区切去一部分分配给请求者, 其依据是当一个很大的空闲区被切割了一部分后可能仍是一个较大的空闲区, 从而避免了空闲区越分越小的问题。这种算法基本不留下小的空闲分区, 但较大的空闲分区也不被保留。

在内存的回收算法上, 当一个任务运行结束, 并释放它所占用的分区后, 如果该分区的左右邻居也是空闲分区, 则需要将它们合并为一个大的空闲分区。与此相对应, 在分区链表上, 也要将相应的链接节点进行合并, 并对其内容进行更新。



### 3. 分区存储管理实例

图 3-48 所示是一个嵌入式系统的内存布局，主要是堆空间的管理。如图 3-48 所示，整个堆空间被划分为两部分：一部分是内存块池（block pools）；一部分是字节池（byte pools）。前者为固定分区的存储管理方式，后者为可变分区的存储管理方式。需要说明的是，这里的分区存储管理讨论的并不是各个任务的地址空间，而是系统的堆空间，即各个任务在运行的时候，如果它们需要使用动态内存，就会通过类似于 malloc 的函数提出申请，系统就会从堆当中分配相应的空间，满足任务的动态内存请求，而不是把整个任务装进来。

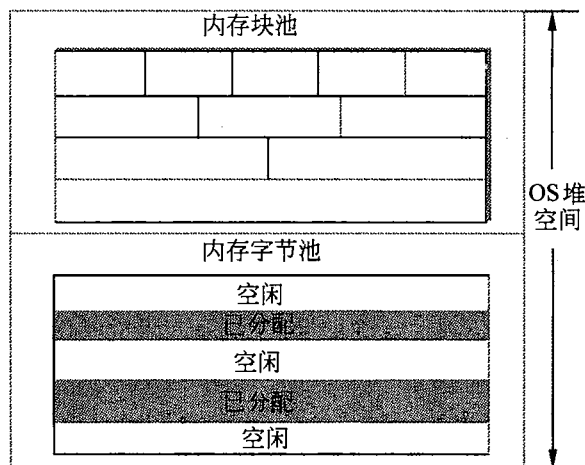


图 3-48 一个嵌入式系统的堆空间

图 3-49（a）所示是内存块池的具体分布。该区域总的大小为 1408KB，被划分为不同大小的块。例如，大小为 64 个字节的块，有 1024 个，共 64KB；大小为 512 个字节的块，有 32 个，共 16KB，等等。最大的块为 512KB，只有一个。另外，当任务去申请一块内存空间时，其大小并不一定与某个块的大小正好相同。例如，假设任务申请的对象大小为 9KB，系统只能将一个 16KB 的块分配给它，所以内碎片为 7KB。

图 3-49（b）所示是字节池的示意图，任务可以申请一块任意大小的内存空间。在当前状态下，如果下一次请求的大小不超过段 1，就会把它一分为二，满足此次请求。另外，虽然在系统中有四块空闲分区，但它们并没有连接在一起。因此，当任务在提出内存申请的时候，能够满足的最大请求为段 5 和段 6 这两块分区之和。实际上，在内存回收算法当中，应该会把这两个相邻的分区进行合并，形成一个大的空闲分区。

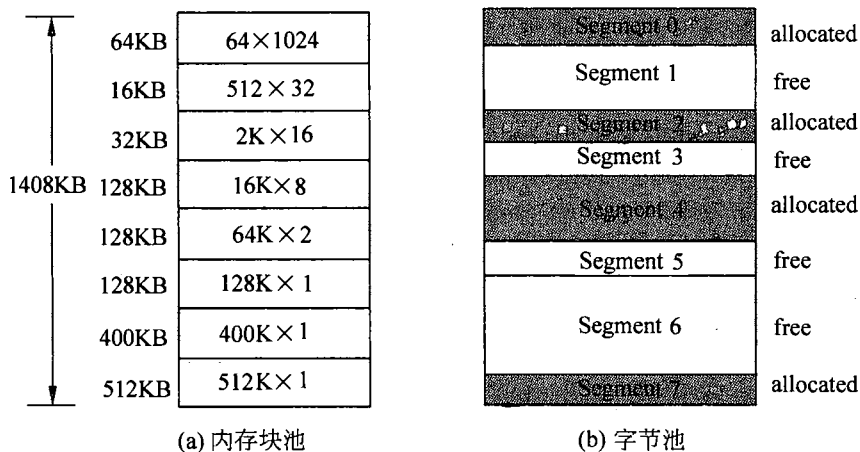


图 3-49 内存块池与字节池

### 3.4.4 地址映射

#### 1. 地址映射概述

地址映射也叫地址重定位，它涉及到两个基本概念，即物理地址和逻辑地址。

- 物理地址 (physical address) 也叫内存地址、绝对地址或实地址。也就是说，把系统内存分割成很多个大小相等的存储单元，如字节或字，每个单元给它一个编号，这个编号就称为物理地址。我们只有通过物理地址，才能对内存单元进行直接访问。物理地址的集合就称为物理地址空间，或者内存地址空间，它是一个一维的线性空间。例如，假设内存大小为 256MB，那么它的内存地址空间是从 0x0 到 0xFFFFFFFF。
- 逻辑地址 (logical address) 也叫相对地址或虚地址。也就是说，用户的程序经过汇编或编译后形成目标代码，而目标代码通常采用的就是相对地址的形式，其首地址为 0，其余指令中的地址都是相对于这个首地址来编址的。显然，逻辑地址和物理地址是完全不同的，我们不能用逻辑地址来直接访问内存单元。

为了保证 CPU 在执行指令的时候，可以正确地访问内存单元，需要将用户程序中的逻辑地址转换为运行时由机器直接寻址的物理地址，这个过程就称为地址映射。

地址映射是由存储管理单元 MMU 来完成的。如图 3-50 所示，当一条指令在 CPU 当中执行时，它可能需要去访问内存，所示就发送一个逻辑地址给 MMU，MMU 负责把这个逻辑地址转换为相应的物理地址，并根据这个物理地址去访问内存。

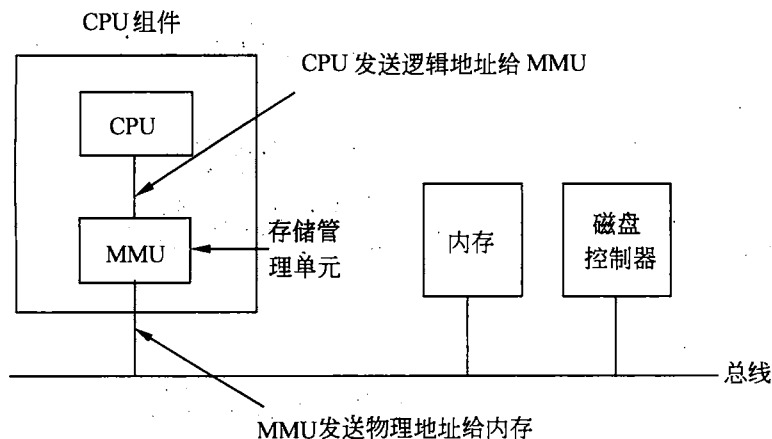


图 3-50 地址映射过程

图 3-51 所示是一个地址映射的例子。图 3-51 (a) 所示是一段简单的 C 语言程序，它首先定义了两个整型变量  $x$  和  $y$ ，然后把  $x$  赋值为 5，再把  $x$  加上 3 并赋值给  $y$ 。经过编译连接后，得到的指令形式类似于图 3-51 (b)。在它的逻辑地址空间中，首地址为 0，代码存放在起始地址为 100 的地方，数据则放在起始地址为 200 的地方，第一条指令 `str 5 [200]`，把常量 5，保存到地址为 200 的地方，这条指令对应于源代码中的  $x=5$ 。也就是说，经过编译和连接后，像  $x$  和  $y$  这样的符号变量都会被具体的逻辑地址所代替， $x$  的逻辑地址是 200， $y$  的逻辑地址是 204。接下来的三条指令，对应于源代码中的  $y=x+3$ 。

假设这个程序即将开始运行，先要把它装入到内存。如果系统采用的是固定分区的存储管理方法，这个程序将被装入到某个空闲分区当中。假设该分区的起始地址是 1000，如图 3-51 (c) 所示，这是装入内存以后的情况。由于程序已经在内存当中，所以现在的地址都是实际的物理地址。但问题立刻就出现了，在程序指令中，它们所采用的地址，还是刚才的逻辑地址，如 200、204，但是 CPU 在执行指令的时候，是按照物理地址来进行的，所以就会把 200 和 204 当成是内存的物理地址去访问了。但这样的话就会出错，因为在物理地址为 200 和 204 的地方，存放的很可能是操作系统的内容，如果对这些内容进行读写操作，可能会对操作系统造成破坏，从而引起系统的崩溃。其实这里的本意并非如此，它实际上是想去访问变量  $x$  和  $y$ ，但它们在内存当中的地址是 1200 和 1204，而不是 200 和 204。另外，如果这个程序被装入另外一个分区，起始地址不是 1000，那么所有的这些地址又都会发生变化。

因此，为了保证 CPU 在执行指令时可以正确地访问存储单元，系统在装入一个用户程序后，必须对它进行地址映射，把程序当中的逻辑地址转换为物理地址，然后才能运行。

地址映射主要有两种方式：静态地址映射和动态地址映射。

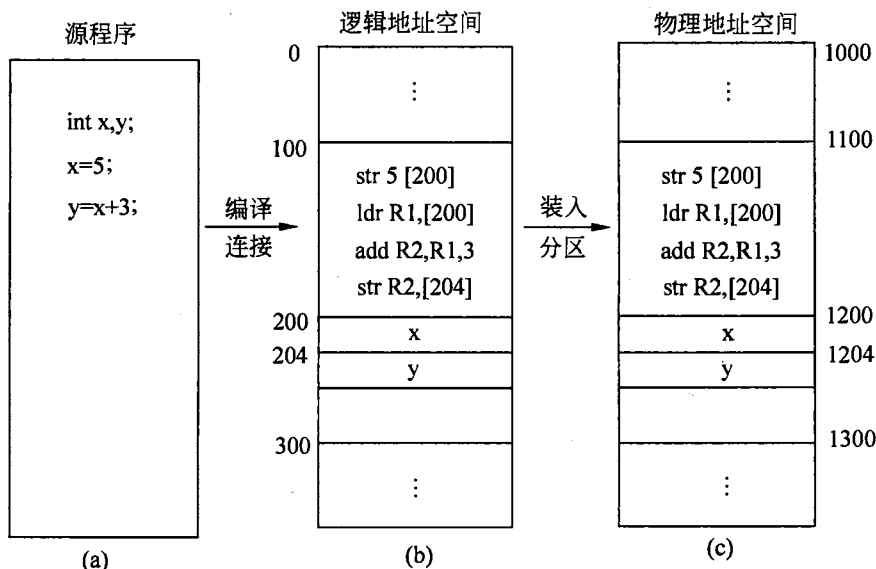


图 3-51 地址映射的例子

## 2. 静态地址映射

静态地址映射的基本思路是：当用户程序被装入内存时，直接对指令代码进行修改，一次性地实现逻辑地址到物理地址的转换。在具体实现时，在每一个可执行文件中，要列出各个需要重定位的地址单元的位置，然后由一个加载程序来完成装入及地址转换的过程。这种方式实现起来很简单，不需要任何硬件方面的支持，但它的缺点是，程序一旦装入到内存以后，就不能再移动。

图 3-52 所示是静态地址映射的一个例子。在装入之前，代码内部使用的是逻辑地址。在装入以后，由于分区的起始地址是1000，所以就把这四条指令当中所有的逻辑地址都进行修改，把它们加上起始地址 1000。从 200 变成了 1200，从 204 变成了 1204。对于第三条指令，它没有访问任何内存单元，所以就不用去修改它。经过这样的修改后，所有的逻辑地址都转换成了物理地址，这一段程序就可以正确地运行了。

## 3. 动态地址映射

动态地址映射的基本思路是：当用户程序被装入内存时，不对指令代码做任何修改，而是在程序的运行过程中，当它需要访问内存单元的时候，再来进行地址转换。在具体实现时，为了提高效率，这项转换工作一般是由硬件的地址映射机制来完成。通常的做法是

设置一个基地址寄存器，或者叫重定位寄存器，当一个任务被调度运行时，就把它所在分区的起始地址装入到这个寄存器中。然后，在程序的运行过程中，当需要访问某个内存单元时，硬件就会自动地将其中的逻辑地址加上基地址寄存器当中的内容，从而得到实际的物理地址，并按照这个物理地址去执行。

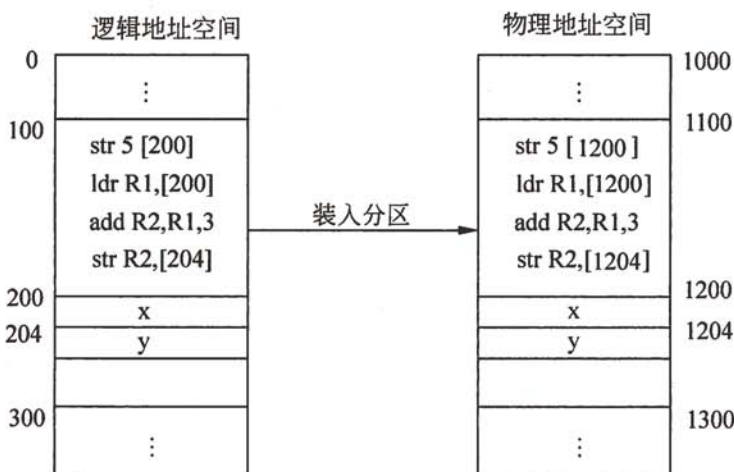


图 3-52 静态地址映射

图 3-53 所示是动态地址映射的一个例子。如图 3-53 所示，当程序在装入内存之前，它里面所用的都是逻辑地址，当它被装入内存后，这些指令代码没有发生任何的变化，里面使用的还是逻辑地址。显然，对于这样的程序，如果直接去运行的话，肯定会出错。但现在新增了一个基地址寄存器。有了它以后，指令的执行方式就发生了变化。例如，在执行第三条指令 `add R2, R1, 3` 时，由于这条指令只涉及到两个 CPU 寄存器，不需要去访问内存单元，所以它的执行方式和原来是完全一样的，没有任何变化。可对于其他指令，如 `str 5 [200]`，它需要去访问逻辑地址 200。而该地址所对应的变量 `x`，已经存放在物理地址 1200 的位置，但 CPU 会自动完成这个转换。当操作系统调度了这个任务去运行时，它所在分区的起始地址，也就是 1000，就会被装入到基地址寄存器当中。然后，当执行到 `str 5 [200]` 这一条指令时，硬件装置就会自动地把其中的相对地址 200 取出来，把它和基地址寄存器做一个加法，从而得到实际的物理地址，也就是 1200，然后再根据这个新的地址来访问内存单元。这样的话，就使得程序能够正确地运行。这个基地址寄存器位于 MMU 的内部，整个地址映射过程是自动进行的。从理论上来说，每访问一次内存都要进行一次地址映射。

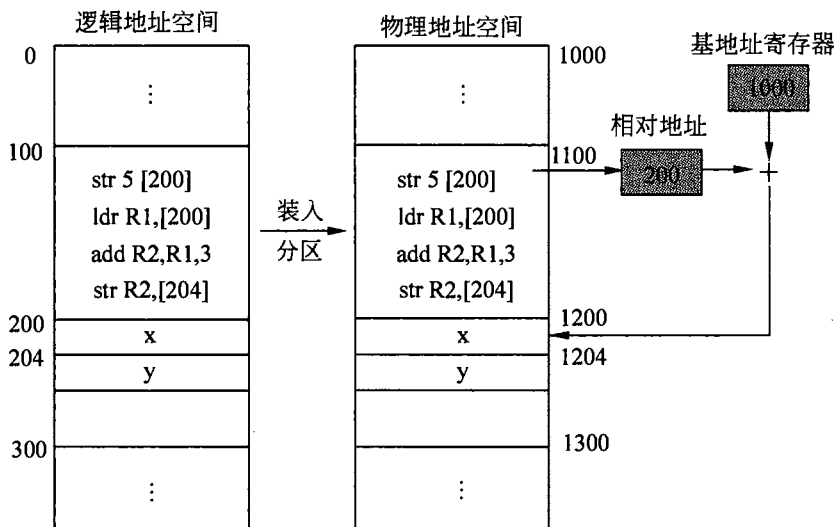


图 3-53 动态地址映射

### 3.4.5 页式存储管理

#### 1. 基本原理

如前所述,分区存储管理的一个特点是连续性,每个程序都分有一片连续的内存区域。这种连续性将导致碎片问题,包括固定分区中的内碎片和可变分区中的外碎片。为了解决这些问题,人们又提出了页式存储管理方案。它的基本出发点是打破存储分配的连续性,使一个程序的逻辑地址空间可以分布在若干个离散的内存块上,从而达到充分利用内存,提高内存利用率的目的。

页式存储管理的基本思路是:一方面,把物理内存划分为许多个固定大小的内存块,称为物理页面 (physical page),或页框 (page frame);另一方面,把逻辑地址空间也划分为大小相同的块,称为逻辑页面 (logical page),或简称为页面 (page),页面的大小要求是  $2^n$ ,一般在 512 个字节到 8KB 之间。当一个用户程序被装入内存时,不是以整个程序为单位,把它存放在一整块连续的区域,而是以页面为单位来进行分配的。对于一个大小为  $N$  个页面的程序,需要有  $N$  个空闲的物理页面把它装进来,当然,这些物理页面不一定是连续的。

图 3-54 所示是一个具体的例子。各个任务的逻辑地址空间和内存的物理地址空间被划分为 1KB 大小的页面。任务 1 有两个页面,任务 2 有三个页面,任务 3 有一个页面。当这三个任务被装入内存后,它们在内存空间的分布可能是:任务 1 的两个页面分别存放在第

五和第六个物理页面中，它们碰巧被放在了一起。任务 2 的三个页面分别存放在第二、第四和第七个物理页面中。也就是说，虽然它们在逻辑地址空间是三个连续的页面，但在物理地址空间却被打散在内存的不同位置。最后，任务 3 的这个页面被存放在第八个物理页面中。

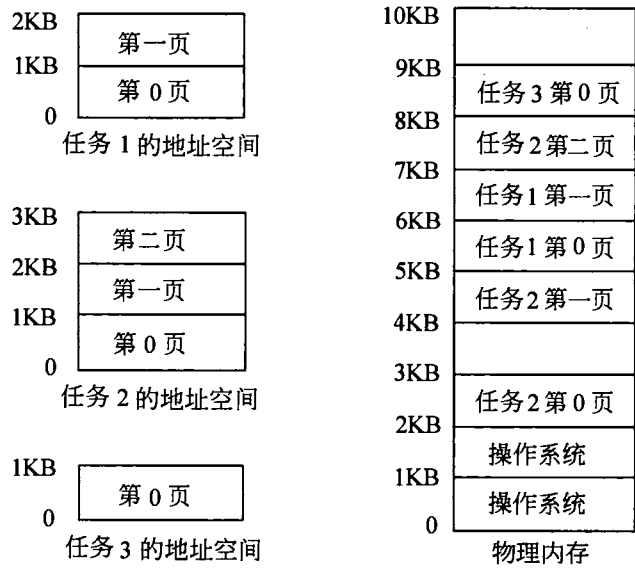


图 3-54 页式存储管理的一个例子

在实现页式存储管理的时候，需要解决以下几个问题。

- 数据结构：用于存储管理的数据结构是什么？
- 内存的分配与回收：当一个任务到来时，如何给它分配内存空间？当一个任务运行结束后，如何来回收它所占用的内存空间？
- 地址映射：当一个任务被加载到内存后，它被打散地存放在若干个不连续的物理页面当中。在这种情况下，如何把程序中使用的逻辑地址转换为内存访问时的物理地址，以确保它能正确地运行？

2. 数据结构

在页式存储管理中，最主要的数据结构有两个。

- 页表 (page table)：页表给出了任务的逻辑页面号与内存中的物理页面号之间的对应关系。

- 物理页面表：用来描述内存空间当中，各个物理页面的使用分配状况。在具体实现上，可以采用位示图或空闲页面链表等方法。

图 3-55 所示是页表的一个例子。在任务的逻辑地址空间当中，总共有四个页面，即页面 0、页面 1、页面 2 和页面 3。页表描述的是逻辑页面号与物理页面号之间的对应关系，即每一个逻辑页面存放在哪一个物理页面中。页表的下标是逻辑页面号，从 0 到 3。相应的页表项存放的就是该逻辑页面所对应的物理页面号。在本例中，任务的四个逻辑页面分别存放在第一、第四、第三和第七个物理页面中。

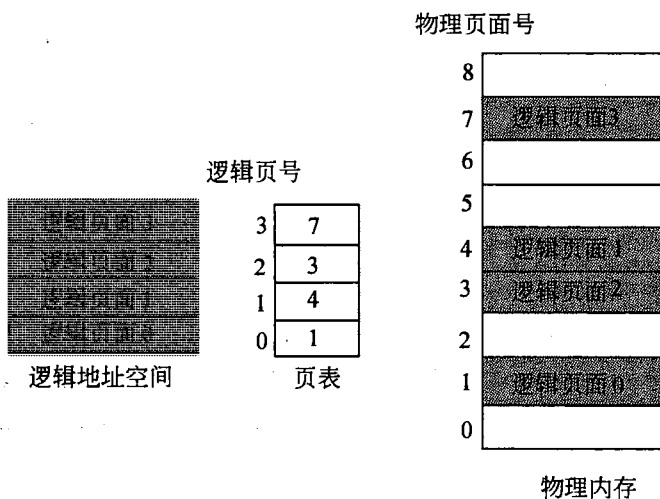


图 3-55 页表的一个例子

### 3. 内存的分配与回收

当一个任务到来时，需要给它分配相应的内存空间，把它的每一个逻辑页面都装入到内存当中。显然，内存的分配与回收算法与物理页面表的实现方法是密切相关的。以位示图为例，内存的分配过程如下：

- (1) 对于一个新来的任务，计算它所需要的页面数  $N$ ，然后查看位示图，看是否还有  $N$  个空闲的物理页面。
- (2) 如果有足够的空闲物理页面，就去申请一个页表，其长度为  $N$ ，并把页表的起始地址填入到该任务的任务控制块 TCB 当中。
- (3) 分配  $N$  个空闲的物理页面，把它们的编号填入到页表中。这样，就建立了逻辑页面与物理页面之间的对应关系。
- (4) 修改位示图，对刚刚被占用的那些物理页面进行标记。



当一个任务运行结束，释放了它所占用的内存空间后，需要对这些物理页面进行回收，并对位示图的内容进行相应修改。

#### 4. 地址映射

如前所述，当一个任务被加载到内存后，它的各个连续的逻辑页面，被打散地存放在若干个不连续的物理页面当中。在这种情况下，为了保证程序能够正确地运行，需要把程序中使用的逻辑地址转换为内存访问时的物理地址，也就是地址映射。

那么如何将一个逻辑地址映射为相应的物理地址呢？我们知道，在页式存储管理当中，连续的逻辑地址空间被划分为一个个的逻辑页面，这些逻辑页面被装入到不同的物理页面当中。也就是说，系统是以页面为单位来进行处理的，而不是以一个个的字节为单位。因此，地址映射的基本思路如下。

- 逻辑地址分析：对于给定的一个逻辑地址，找到它所在的逻辑页面，以及它在页面内的偏移地址。
- 页表查找：根据逻辑页面号，从页表中找到它所对应的物理页面号。
- 物理地址合成：根据物理页面号及页内偏移地址，确定最终的物理地址。

##### (1) 逻辑地址分析

由于页面的大小一般都是 2 的整数次幂，所以，人们可以很方便地进行逻辑地址的分析。具体来说，对于给定的一个逻辑地址，可以直接把它的高位部分作为逻辑页面号，把它的低位部分作为页内偏移地址。例如，假设页面的大小是 4KB，即  $2^{12}$ ，逻辑地址为 32 位，那么在一个逻辑地址当中，最低的 12 位就是页内偏移地址，而剩下的 20 位就是逻辑页面号。

图 3-56 所示是逻辑地址分析的一个例子，在这个例子中，逻辑地址是用十六进制的形式来表示的。假设页面的大小为 1KB，逻辑地址为 0x3BAD。在这种情况下，首先把这个十六进制的地址展开为二进制的形式。然后，由于页面的大小为 1KB，即  $2^{10}$ ，所以这个逻辑地址的最低 10 位，就表示页内偏移地址，而剩下的最高 6 位，就表示逻辑页面号。因此，逻辑页面号是 0x0E，页内偏移地址是 0x03AD。

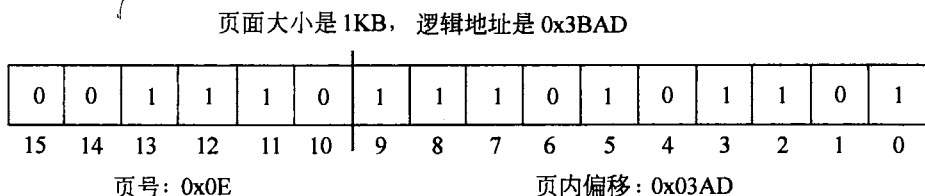


图 3-56 逻辑地址分析的例子

如果逻辑地址不是用十六进制，而是用十进制的形式来表示的，那么有两种做法：一是先把它转换为十六进制的形式，然后重复刚才的步骤；二是采用如下的计算方法：

逻辑页面号 = 逻辑地址 / 页面大小

页内偏移量 = 逻辑地址 % 页面大小

把逻辑地址去除以页面大小，得到的商就是逻辑页面号，得到的余数就是页内偏移地址。例如，假设页面的大小为 2KB，现在要计算逻辑地址 7145 的逻辑页面号和页内偏移地址。把 7145 去除以 2048，得到的商是 3，余数是 1001，所以这个逻辑地址的逻辑页面号是 3，页内偏移地址是 1001。实际上，这个算法同刚才的十六进制的方法是完全等价的。从二进制运算的角度来看，一个是右移操作，一个是除法操作。这两个操作的效果是等价的：把一个整数右移  $N$  位等价于把它除以  $2^N$  次方。

### (2) 页表查找

对于给定的一个逻辑地址，如果我们已经知道了它的逻辑页面号，就可以去查找页表，从中找到相应的物理页面号。

在具体实现上，页表通常保存在内核的地址空间中，因为它是操作系统的一个数据结构。另外，为了能够访问页表的内容，在硬件上要增加一对寄存器：一个是页表基地址寄存器，用来指向页表的起始地址；另一个是页表长度寄存器，用来指示页表的大小，即对于当前任务，它总共包含有多少个页面。操作系统在进行任务切换的时候，会去更新这两个寄存器当中的内容。

### (3) 物理地址合成

对于给定的一个逻辑地址，如果已经知道了它所对应的物理页面号和页内偏移地址，那么可以采用简单的叠加算法，计算出最终的物理地址。例如，假设物理页面号为  $f$ ，页内偏移地址为  $offset$ ，每个页面的大小为  $2^n$ ，那么相应的物理地址为  $f \times 2^n + offset$ 。

图 3-57 所示是页式存储管理当中的地址映射机制，也是以上各个步骤的一个综合。假设在程序的运行过程中，需要去访问某个内存单元，所以就给出了这个内存单元的逻辑地址。如前所述，这个逻辑地址由两部分组成：一是逻辑页面号，二是页内偏移地址。这个分析工作是由硬件自动来完成的，对用户是透明的。在页表基地址寄存器当中，存放的是当前任务的页表首地址。将这个首地址与逻辑页面号相加，就找到了相应的页表项，里面存放的是这个逻辑页面所对应的物理页面号。将这个物理页面号取出来，与页内偏移地址进行组合，从而得到最终的物理地址，然后就可以用这个物理地址去访问内存。

现有的这种地址映射方案，虽然能够实现从逻辑地址到物理地址的转换，但它有一个很大的问题。当程序在运行时，如果需要去访问某个内存单元，例如，去读写内存其中的一个数据，或是去内存取一条指令，在此情况下，需要访问两次内存：第一次是去访问页表，取出物理页面号；第二次才是真正去访问数据或指令。也就是说，内存的访问效率只

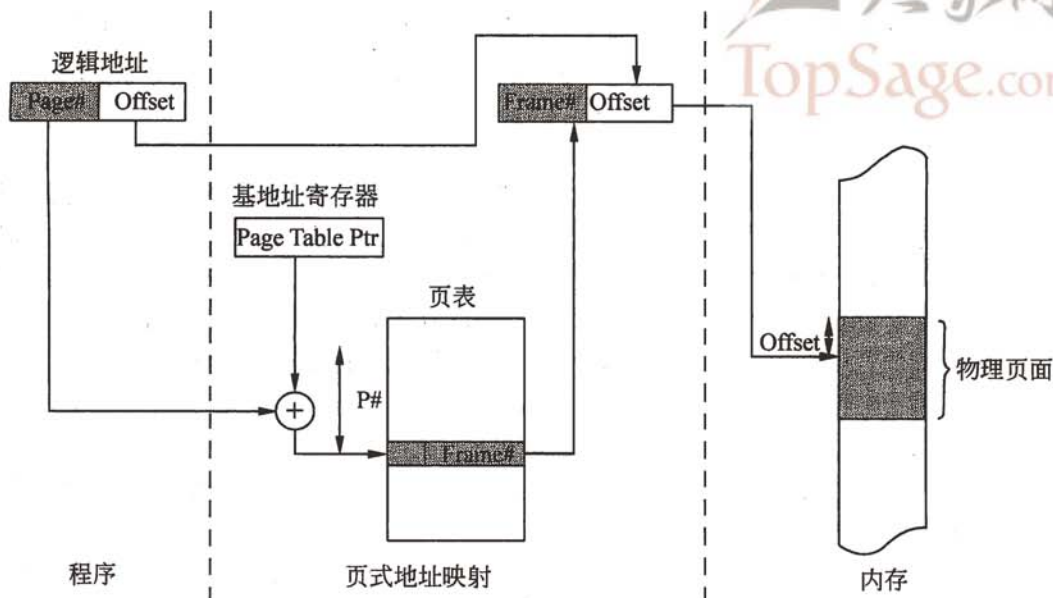


图 3-57 页式存储管理中的地址映射

有 50%。这样，就会降低内存的存取速度，进而影响到整个系统的使用效率。为了解决这个问题，人们又引入了快表的概念。它的基本思路来源于对程序运行过程的一个观察结果，也就是说，对于绝大多数的程序来说，它们在运行的时候，倾向于集中访问一小部分的页面。因此，对于它们的页表来说，在一定时间内，只有一小部分的页表项会被经常访问，而其他的页表项则很少使用。根据这个观察结果，人们在 MMU 中增加了一种特殊的快速查找硬件——TLB (Translation Lookaside Buffer)，或者叫关联存储器，用来存放那些最常用的页表项。这种硬件设备能够把逻辑页面号直接映射为相应的物理页面号，而不需要再去访问内存当中的页表，这样就缩短了页表的查找时间。

在 TLB 方式下，地址映射的过程略有不同。当一个逻辑地址到来时，它首先会到 TLB 当中去查找，看这个逻辑页面号所在的页表项是否包含在 TLB 当中。这个查找的速度是非常快的，因为它是以并行的方式进行的。如果能够找到的话，就直接地从 TLB 当中把相应的物理页面号取出来，与页内偏移地址拼接成最终的物理地址；如果在 TLB 当中没有找到该逻辑页面，那只能采用通常的地址映射方法，去访问内存当中的页表。接下来，硬件还会在 TLB 当中寻找一个空闲单元，如果没有空闲单元，就把某一个页表项驱逐出来，然后把刚刚访问过的这个页表项添加到 TLB 当中。这样，如果下次再来访问这个页面，就能在

TLB 当中找到。

### 5. 优缺点

页式存储管理方案的优点是：

- 没有外碎片，而且内碎片的大小不会超过页面的大小。这是因为系统以页面来作为内存分配的基本单位，每一个页面都能够用上，不会浪费，只是在任务的某一些页面当中，可能没有装满，里面有一些内碎片。
- 程序不必连续存放，它可以分散地存放在内存的不同位置，从而提高了内存利用率。
- 便于管理。

它的缺点主要有：

- 程序必须全部装入内存，才能够运行。如果一个程序的规模大于当前的空闲空间的总和，那么它就无法运行。
- 操作系统必须为每一个任务都维护一张页表，开销比较大，简单的页表结构已经不能满足要求，必须设计出更为复杂的结构，如多级页表结构、哈希页表结构、反置页表等。

## 3.4.6 虚拟存储管理

在操作系统的支持下，MMU 还提供虚拟存储功能，即使一个任务所需要的内存空间超过了系统所能提供的内存空间，也能够正常运行。

### 1. 程序局部性原理

程序的局部性原理，指的是程序在执行过程中的一个较短时期内，它所执行的指令和访问的存储空间分别局限在一定的区域内。这可以表现在时间和空间两个方面。

- 时间局限性：一条指令的一次执行和下一次执行、一个数据的一次访问和下一次访问，都集中在一个较短的时期内。
- 空间局限性：如果程序执行了某条指令，则它相邻的几条指令也可能马上被执行；如果程序访问了某个数据，则它相邻的几个数据也可能马上被访问。

程序局部性原理的具体表现：

- 程序在执行时，大部分都是顺序执行的指令，只有少部分是跳转和函数调用指令。而顺序执行就意味着在一小段时间内，CPU 所执行的若干条指令在地址空间当中是连续的，集中在一个很小的区域内。
- 程序中存在相当多的循环结构，在这些循环结构的循环体当中，只有少量的指令，它们会被多次执行。
- 程序中存在相当多对一定数据结构的操作，这些操作往往局限在比较小的范围内。例如数组操作，数组是连续分配的，各个数组元素之间是相邻的。

程序的局部性原理说明, 在一个程序的运行过程中, 在某一段时间内, 这个程序只有一小部分的内容是处于活跃状态的, 正在被使用, 而其他的大部分内容可能都处于一种休眠状态, 没有被使用, 而这就意味着, 从理论上来说, 虚拟存储技术是能够实现的, 而且在实现了以后, 应该能够取得一个满意的效果。实际上, 在很多地方, 都已经用到了程序的局部性原理, 例如, 页式地址映射当中的 TLB、CPU 里面的 Cache 等, 都是基于局部性原理。

## 2. 虚拟页式存储管理

虚拟页式存储管理就是在页式存储管理的基础上, 增加了请求调页和页面置换的功能。它的基本思路是: 当一个用户程序需要调入内存去运行时, 不是将这个程序的所有页面都装入内存, 而是只装入部分的页面, 就可以启动这个程序去运行。在运行过程中, 如果发现要执行的指令或者要访问的数据不在内存当中, 就向系统发出缺页中断请求, 然后系统在处理这个中断请求时, 就会将保存在外存中的相应页面调入内存, 从而使该程序能够继续运行。

在单纯的页式存储管理当中, 页表的功能就是把逻辑页面号映射为相应的物理页面号。因此, 对于每一个页表项来说, 它只需要两个信息: 一个是逻辑页面号, 另一个是与之相对应的物理页面号。但是在虚拟页式存储管理当中, 除了这两个信息之外, 还要增加其他一些信息, 包括驻留位、保护位、修改位和访问位。

- 驻留位: 表示这个页面现在是在内存还是在外存。如果这一位等于1, 表示页面位于内存中, 即页表项是有效的; 如果这一位等于0, 表示页面还在外存中, 即页表项是无效的, 如果此时去访问, 将会导致缺页中断。
- 保护位: 表示允许对这个页面做何种类型的访问, 如只读、可读写、可执行等。
- 修改位: 表示这个页面是否曾经被修改过。如果该页面的内容被修改过, CPU 会自动把这一位的值设置为 1。
- 访问位: 如果这个页面曾经被访问过, 包括读操作、写操作等, 那么这一位就会被硬件设置为 1。这个信息主要用在页面置换算法当中。

图 3-58 所示是虚拟页式存储管理方式下, 地址映射的一个例子。在这个例子中, 逻辑地址是 16 位, 逻辑地址空间的大小是 64KB。而对于实际的物理内存来说, 它的大小只有 32KB。假设每个页面的大小是 4KB, 这样的话, 逻辑地址空间总共有 16 个逻辑页面, 编号从 0~15。而对于物理内存来说, 总共有八个物理页面, 编号从 0~7。在页表当中, 记录了每一个逻辑页面所对应的物理页面号, 但由于逻辑地址空间远远大于实际的内存空间, 所以不可能把所有的逻辑页面都装入到内存当中。如图 3-58 所示, 只有八个逻辑页面被装入到内存, 相应的, 在它们的页表项当中, 就包含有相对应的物理页面号, 而且驻留位的值是 1。而对于剩下的八个逻辑页面, 它们被保存在外存当中, 所以在它们的页表项当中, 驻留位的值就等于 0, 表示没有相对应的物理页面号, 因此就用一个 X 来表示。假设现在要去执行一条指令: `MOV REG,[0]`, 也就是说, 把逻辑地址 0 当中的值取出来, 放到寄存

器 REG 当中。因此, CPU 就会把逻辑地址 0 发送给 MMU, MMU 首先对它进行分析, 得出它的逻辑页面号为 0, 页内偏移地址为 0。然后查找页表, 发现该页面存放在第二个物理页面当中, 这样就可以计算出最后的物理地址为  $2 \times 4\text{KB} + 0$ , 即 8192。最后, MMU 就会把这个物理地址发送到地址总线上, 从而去访问相应的内存单元。这样, 这一次内存访问就顺利地完成了。

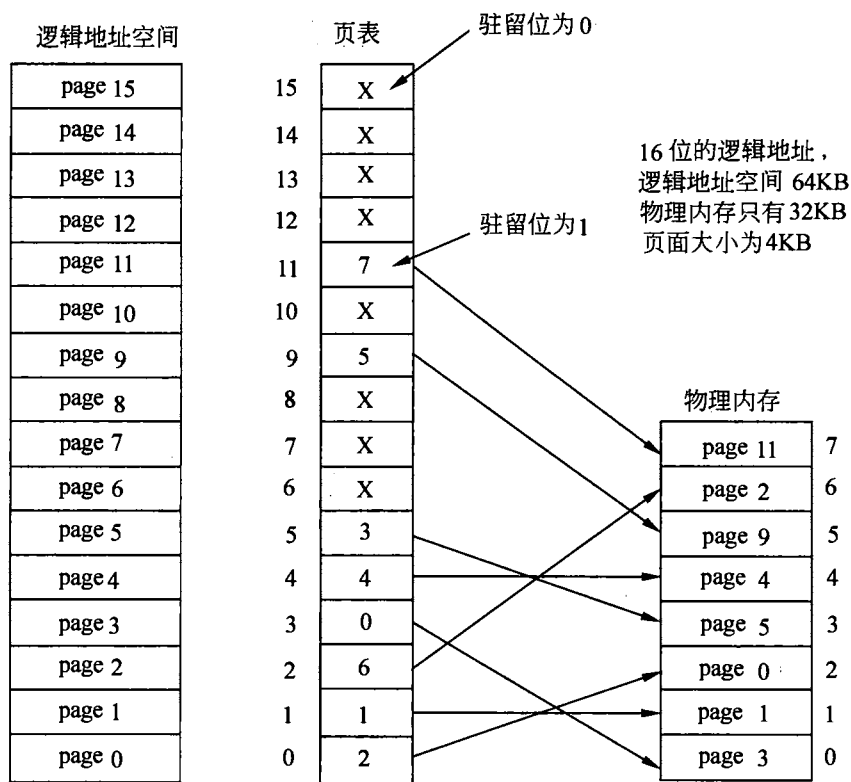


图 3-58 虚拟页式存储管理中的地址映射

假设现在要去执行另外一条指令 `MOV REG, [32780]`, 也就是对逻辑地址 32780 进行访问。MMU 会发现这个逻辑地址是位于第八个逻辑页面当中, 页内偏移地址为 12, 所以它就用 8 作为索引, 到页表中去查找。结果发现在第八个逻辑页面的页表项当中, 驻留位的值等于 0, 也就是说, 这个页面并没有在内存当中。这样的话, MMU 就没有办法继续下去, 只能引发一个缺页中断, 把这个问题提交给操作系统去处理。

当一个缺页中断发生时, 操作系统是如何来处理的呢? 如图 3-59 所示, 我们把中断的处理过程用一个流程图来表示。当发生一个缺页中断时, 首先判断在内存中是否还有空闲

的物理页面。如果有，就分配一个空闲页面出来；如果没有，就要采用某种页面置换算法，从内存当中，选择一个即将被替换出去的页面。对这个页面的处理也要分两种情况。如果这个页面在内存期间曾经被修改过，也就是说，在它的页表项里面，修改位的值等于 1，那么就把它的内容写回到外存当中；如果这个页面在内存期间没有被修改过，那么就什么都不用做，到时候它自然而然会被新的页面所覆盖。现在我们已经有了一个可供使用的物理页面，不管这个页面是直接分配的空闲页面，还是将某个页面置换出去后空出来的，接下来，就可以把需要访问的新的逻辑页面装入到这个物理页面当中，并修改相应的页表项的内容，包括驻留位、物理页面号等。最后退出中断，重新运行中断前的指令。当这条指令重新运行的时候，由于它需要访问的逻辑页面已经在内存当中，所以就能够顺利地运行下去，不会再产生缺页中断。

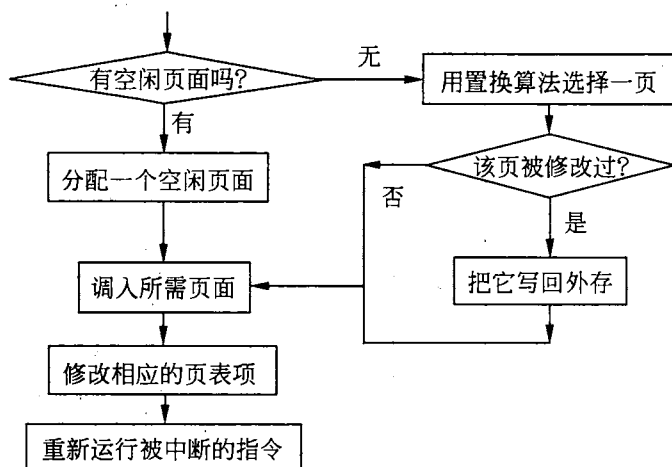


图 3-59 缺页中断的处理过程

回到刚才的那个例子，假设现在要去执行指令 `MOV REG, [32780]`，对逻辑地址 32780 进行访问。这个逻辑地址位于第八个逻辑页面当中，页内偏移地址为 12。如图 3-58 所示，在查找页表的时候发现，这个逻辑页面并没有被调入内存，而是保存在外存当中，这样就会引发一个缺页中断。在中断处理过程中，系统会首先判断在内存中是否存在有空闲的物理页面，结果发现八个物理页面已经全部分配出去了。因此，只好采用某种页面置换算法，去选择一个页面，把它置换出去。假设选中的是第一个逻辑页面，那么系统会查看这个页面在内存期间是否曾经被修改过，如果是，就把它的内容写回到外存，这样，原来用来存放它的第一个物理页面就空闲下来了。接下来，就可以把所需的新的逻辑页面装入到第一个物理页面当中。然后，我们还需要去修改相应的页表项的内容。首先修改逻辑页面 1 所

对应的页表项，把驻留位的值设置为 0，使它不再指向任何物理页面，然后再修改第八个逻辑页面的页表项，把它的驻留位设置为 1，并且把它的物理页面号设置为 1。

### 3. 页面置换算法

如前所述，系统在处理缺页中断时，需要调入新的页面。如果此时内存已满，就要采用某种页面置换算法，从内存中选择某一个页面，把它置换出去。最简单的做法是随机地进行选择，但这显然不是一个令人满意的方法。例如，假设随机选中的是一个经常要访问的页面，当它被置换出去后，可能马上又得把它换进来，而这种换进换出是需要开销的。因此，对于一个好的页面置换算法来说，它应该尽可能多地减少页面的换进换出次数，或者说，尽可能多地减少缺页中断的次数，从而减少系统在这方面的开销。具体来说，它应该把那些将来不再使用的，或者短期内较少使用的页面换出去，而把那些经常要访问的页面保留下来。不过，在通常情况下，我们不可能完全做到这一点，因为在任意一个时刻，我们只可能知道过去已经发生的访问情况，而不知道未来即将要发生的访问情况——我们无法确切地知道，在不远的将来，哪些页面会被频繁访问，哪些页面不会被访问。因此，通常的做法，就是在程序局部性原理的指导下，依据过去的统计数据来对将来进行预测。常用的页面置换算法包括：最优页面置换算法、最近最久未使用算法、最不常用算法、先进先出算法和时钟页面置换算法。

#### (1) 最优页面置换算法 (OPTimal, OPT)

最优页面置换算法的基本思路是：当一个缺页中断发生时，对于内存当中的每一个逻辑页面，计算在它的下一次访问之前，还要等待多长的时间，然后从中选择等待时间最长的那个，来作为被置换的页面。从算法本身来看，这的确是一个最优算法，它能保证缺页中断的发生次数是最少的。但是，这个算法只是一种理想化的算法，在实际的系统中是无法实现的，因为操作系统无从知道，每一个页面还要等待多长的时间才会被再次地访问。因此，OPT 算法通常用作其他算法的性能评价依据。

#### (2) 最近最久未使用算法 (Least Recently Used, LRU)

最近最久未使用算法的基本思路是：当一个缺页中断发生时，从内存中选择最近最久没有被使用的那个页面，把它淘汰出局。LRU 算法实质上是对最优页面置换算法的一个近似，它的理论依据就是程序的局部性原理。也就是说，如果在最近一小段时间内，某些页面被频繁地访问，那么在将来的一小段时间内，这些页面可能会再次被频繁地访问。反之，如果在过去一段时间内，某些页面长时间没有被访问，那么在将来，它们还可能会长时间得不到访问。OPT 算法寻找的是将来长时间内得不到访问的那个页面，而 LRU 算法寻找的是过去长时间内没有被访问的那个页面。

LRU 算法需要记录各个页面在使用时间上的先后顺序，所以系统的开销比较大。在具体实现上，主要有两种方法。



- 链表法：由系统来维护一个页面链表，把最近刚刚使用过的页面作为首节点，把最久没有使用的页面作为尾节点。在每一次访问内存的时候，找到相应的逻辑页面，把它从链表中摘下来，移动到链表的开头，成为新的首节点。然后，当发生缺页中断的时候，总是淘汰链表末尾的那个页面，因为它就是最久未使用的。
- 栈方法：由系统来设置一个页面栈，每当访问一个逻辑页面时，就把相应的页面号压入到栈顶，然后考察栈内是否有与之相同的页面号，如果有就把它抽出来。当需要淘汰一个页面时，总是选择栈底的页面，因为它就是最久未使用的。

### (3) 最不常用算法 (Least Frequently Used, LFU)

最不常用算法的基本思路是：当一个缺页中断发生时，选择访问次数最少的那个页面，把它淘汰出局。在具体实现上，需要对每一个页面都设置一个访问计数器，每当一个页面被访问时，就把它的计数器的值加 1。然后在发生缺页中断的时候，选择计数值最小的那个页面，把它置换出去。LFU 算法和 LRU 算法类似，都是基于程序的局部性原理，通过分析过去的访问情况来预测将来的访问情况。两者的区别在于：LRU 考察的是访问的时间间隔，即对于每一个页面，从它的上一次访问到现在，经历了多长的时间；而 LFU 考察的是访问的频度，即对于每一个页面，在最近一段时间内，它总共被访问了多少次。

### (4) 先进先出算法 (First In First Out, FIFO)

先进先出算法的基本思路是：选择在内存中驻留时间最长的页面，把它淘汰出局。在具体实现上，系统会维护一个链表，里面记录了内存当中的所有页面。从链表的排列顺序来看，链首页面的驻留时间最长，链尾页面的驻留时间最短。当发生一个缺页中断时，把链首的页面淘汰出局，然后把新来的页面添加到链表的末尾。FIFO 算法的性能比较差，因为它每次淘汰的都是驻留时间最长的页面，而这样的页面并不一定就是不常用的页面，相反，可能是一些经常要访问的页面，如果现在把它们置换出去，将来可能还要把它们再换回来。

### (5) 时钟页面置换算法 (Clock)

FIFO 算法的缺点在于：它是根据页面的驻留时间来做出选择，而没有去考虑页面的访问情况。时钟页面置换算法对此进行了改进，把页面的访问情况也作为淘汰页面的一个依据。Clock 算法需要用到页表项当中的访问位。当一个页面在内存当中的时候，如果它被访问了（不管是读操作还是写操作），那么它的访问位就会被 CPU 设置为 1。算法的基本思路是：把各个页面组织成环形链表的形式，类似于一个时钟的表面，然后把指针指向最古老的那个页面，即最先进来的那个页面。当发生一个缺页中断的时候，考察指针所指向的那个页面。如果它的访问位的值等于 0，说明这个页面的驻留时间最长，而且没有被访问过，所以理所当然地把它淘汰出局；如果访问位的值等于 1，则说明这个页面的驻留时间虽然是最长的，但是在这一段时间内，它曾经被访问过了，因此，在这种情况下，就暂

不淘汰这个页面，但要把它访问位的值设置为 0，然后把指针往下移动一格，去考察下一个页面。就这样一直进展下去，直到发现某一个页面它的访问位的值等于 0，就把它淘汰掉。

#### 4. 工作集模型

工作集模型是计算机科学家 Denning 在 20 世纪 60 年代提出来的，它描述的是一个程序在运行过程当中的行为规律。所谓的工作集 (working set)，就是指任务当前正在使用的逻辑页面的集合。它可以用一个二元函数  $W(t, \Delta)$  来表示，其中  $t$  指的是当前的执行时刻， $\Delta$  称为是工作集窗口，也就是一个定长的页面访问窗口。 $W(t, \Delta)$  就等于在  $t$  时刻之前的  $\Delta$  窗口当中，所有页面所组成的集合。显然，随着当前时刻  $t$  的不断变化，任务的工作集也在不断地变化。

图 3-60 所示是工作集的一个例子。假设在某一段时间内，任务所访问的逻辑页面的顺序如图 3-60 所示。如果  $\Delta$  窗口的长度为 10，那么在  $t_1$  时刻，任务的工作集是这样计算的：从  $t_1$  时刻开始，往回数 10 个页面，这 10 个页面就是它的工作集窗口，然后把这个窗口里面的重复页面去除，就得到了任务在  $t_1$  时刻的工作集  $\{1, 2, 5, 6, 7\}$ 。采用类似的方法，可以计算出在  $t_2$  时刻，任务的工作集为  $\{3, 4\}$ 。

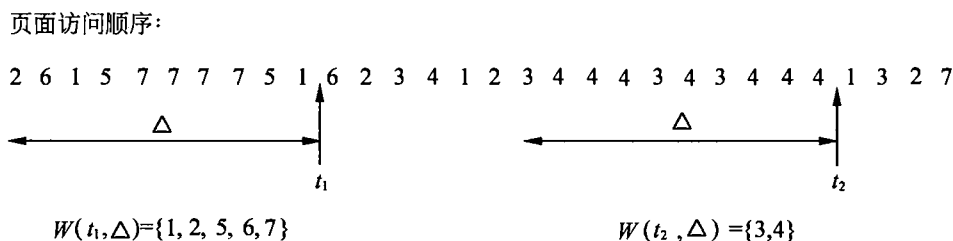


图 3-60 工作集的一个例子

在一个任务的运行过程中，它的工作集是在不断变化的。一般来说，当一个任务刚刚启动时，它会不断地去访问一些新的页面，然后逐步建立一个比较稳定的工作集。当内存访问的局部性区域的位置大致稳定时，工作集的大小也就大致地稳定下来。然后，当内存访问的局部性区域的位置发生改变时，工作集就会快速地扩张和收缩，并且过渡到下一个稳定值。在这些稳定阶段，任务在运行的时候，只会去访问一些固定的页面，而其他的页面一般是不会去访问的，这就是程序的局部性原理的具体表现。因此当一个任务在运行的时候，并不需要把它的整个程序都装入到内存，只要把它的工作集装入到内存中就可以了。

与工作集相关的另一个概念是驻留集。所谓的驻留集，就是在当前时刻，任务实际驻留在内存当中的页面集合。驻留集与工作集既有区别，也有联系。工作集是任务在运行过

程中所固有的性质，而驻留集则取决于系统分配给任务的物理页面个数，以及所采用的页面置换算法。当一个任务在运行时，如果它的整个工作集都在内存当中，也就是说，它的工作集是驻留集的一个子集，那么这个任务将会很顺利地运行，不会造成太多的缺页中断。反之，如果分配给一个任务的物理页面数太少，不能包含整个的工作集，也就是说，驻留集是工作集的一个真子集，在这种情况下，任务将会造成很多缺页中断，需要频繁地进行页面置换，从而使任务的运行速度变得非常慢，这种现象称为“抖动”（thrashing）。

## 3.5 设备管理

### 3.5.1 设备管理基础

在嵌入式系统中，存在着各种类型的输入/输出（I/O）设备，既包括键盘、触摸屏、液晶显示器等人机交互设备，也包括 D/A、A/D 转换器、电机等专用设备。

一个 I/O 单元通常是由两个部分组成的：一个是机械部分，即 I/O 设备本身；另一个是电子部分，即设备控制器或设备适配器。它的功能是完成设备与主机之间的连接和通信。也就是说，I/O 设备本身并不直接与 CPU 打交道，而是通过它的设备控制器来与 CPU 打交道。在每个设备控制器当中，都会有一些寄存器，用来与 CPU 进行通信，包括控制寄存器、状态寄存器和数据寄存器等。通过往这些寄存器当中写入不同的值，操作系统就可以命令设备去执行发送数据、接收数据、打开、关闭等各种操作。另外，操作系统也可以通过读取某些寄存器的值，来了解这个设备的当前状态。

那么 CPU 如何来访问设备控制器当中的这些寄存器呢？如果是访问普通的内存单元，那么很简单，只要指明这个内存单元的地址即可。但是现在要访问的是一些硬件寄存器，所以必须设计出相应的解决方法，这主要有三种：I/O 独立编址、内存映像编址和混合编址。

#### （1）I/O 独立编址

I/O 独立编址的基本思路是：对于各种设备控制器当中的每一个寄存器，分配一个唯一的 I/O 端口编号，也叫 I/O 端口地址，然后用专门的 I/O 指令来对这些端口进行操作。这些端口地址所构成的地址空间是完全独立的，与内存的地址空间没有任何关系。采用这种独立编址的方法，其优点是：I/O 设备不会去占用内存的地址空间，而且在编写程序的时候，很容易区分内存访问和 I/O 端口访问，因为对于不同的操作来说，它们的指令形式是不一样的。

#### （2）内存映像编址

内存映像编址的基本思路是：把各种设备控制器当中的每一个寄存器都映射为一个内存单元，这些内存单元专门用于 I/O 操作，而不能作为普通的内存单元来使用，不能往里

面存放一些与 I/O 无关的数据。不过,从操作的层面上来说,对这些内存单元的读写方式与平常的内存访问是完全相同的,没有任何区别。采用这种内存映像编址的方法,端口地址空间与内存地址空间是统一编址的,端口地址空间是内存地址空间的一部分,而且编程非常方便,无须专门的 I/O 指令。

### (3) 混合编址

混合编址的基本思路就是把以上两种编址方法混合在一起。具体来说,对于设备控制器当中的寄存器,采用独立编址的方法,每一个寄存器都有一个独立的 I/O 端口地址;而对于设备的数据缓冲区,则采用内存映像编址的方法,把它们的地址统一到内存地址空间当中。

## 3.5.2 I/O 控制方式

I/O 设备的控制方式主要有三种:程序循环检测、中断驱动和直接内存访问。

### 1. 程序循环检测方式

程序循环检测方式的基本思路是:在程序(一般是设备驱动程序)当中,通过不断地检测 I/O 设备的当前状态,来控制一个 I/O 操作的完成。具体来说,在进行 I/O 操作之前,要循环地去检测该设备是否已经就绪。如果是,就向控制器发出一条命令,启动这一次的 I/O 操作。然后,在这个操作的进行过程中,也要循环地去检测设备的当前状态,看它是否已经完成。总之,在 I/O 操作的整个过程中,控制 I/O 设备的所有工作都是由 CPU 来完成的。这种方式也称为是繁忙等待方式或轮询方式。它的缺点主要是:在进行一个 I/O 操作的时候,要一直占用着 CPU,这样就会浪费 CPU 的时间。

图 3-61 所示是循环检测方式的一个例子。假设 I/O 地址采用的是内存映像编址方式,现在需要在打印机上打印一个字符串“ABCDEFGH”。对于操作系统来说,要完成这个任务,其实很简单,只要把这八个字符一个接一个地送到打印机设备的 I/O 端口地址就可以了。如图 3-61(a)所示,这八个字符被保存在系统内核的一个缓冲区当中,并用指针 `p` 来指向它们。`status_reg` 这个内存单元对应于打印机控制器里面的状态寄存器, `data_register` 这个内存单元对应于它的数据寄存器,现在要做的事情,就是把这八个字符一个接一个地放到数据寄存器当中。

图 3-61(b)所示是相应的程序。它的基本思路是:逐个去打印每一个字符。在打印一个字符之前,首先用一个 `while` 语句来检测打印机的当前状态,看它是否已经就绪,如果还没有就绪,就在这里循环等待;如果已经就绪,就把当前的字符送入到打印机的数据寄存器当中。在本例中,由于采用了内存映像的编址方式,因此,在程序员眼中,状态寄存器和数据寄存器都被看成是普通的内存单元,对它们的访问也是普通的赋值操作,不需要专门的 I/O 指令。但是这个赋值操作的功能与普通的赋值操作不同,它相当于是给打印

机发出了一个命令，让它去打印一个字符。另外，每次打印完一个字符后，都要重新判断设备是否就绪，因为相对于 CPU 来说，打印机是一个慢速设备，它在执行打印命令时，不可能像 CPU 那么快，而是需要一定的时间来完成。因此，当 CPU 把一个字符交给它之后，必须循环等待一段时间，才能去处理下一个字符。

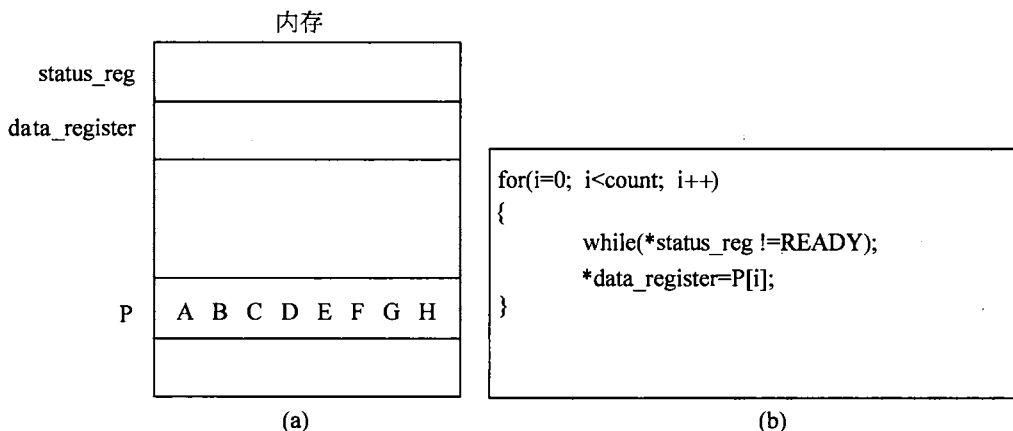


图 3-61 程序循环检测方式的例子

## 2. 中断驱动方式

循环检测的控制方式，需要占用大量的 CPU 时间。例如，假设打印机的打印速度为 100 字符/秒，在循环检测方式下，当一个字符被写入到打印机的数据寄存器后，CPU 要等待 10ms 才能把下一个字符写进去，而这 10ms 的时间，就在循环等待中被白白浪费掉了。为了解决这个问题，一种办法就是让 CPU 在这 10ms 的时间内，先去运行其他的任务，然后等打印机处理完上一个字符后，CPU 再接着处理下一个字符。这种方法被称为是中断驱动的控制方式。它的基本思路是：当一个用户任务需要进行 I/O 操作时，会去调用相应的系统函数，由这个函数来发起 I/O 操作，并将当前任务阻塞起来，然后调度其他的任务去使用 CPU。当所需的 I/O 操作完成时，相应的设备就会向 CPU 发出一个中断，系统在中断处理程序当中，如果发现还有数据需要处理，就再次启动 I/O 操作。在中断驱动的控制方式下，数据的每一次读写还是通过 CPU 来完成，只不过当 I/O 设备在进行数据处理时，CPU 不必在那里等待，而是可以去执行其他任务。

仍以打印字符的问题为例。如图 3-62 所示，在中断驱动方式下，对于用户程序来说，它所做的事情可能是：把需要打印的字符串放到一个缓冲区 buffer 中，然后调用一个系统调用函数 print。在 print 系统调用中，首先把用户缓冲区中的字符串复制到系统内核的字符数组 p 当中，然后打开中断。接下来是一个循环检测语句，判断打印机的当前状态是否

就绪，当打印机就绪后，就把第一个字符放到数据寄存器里面去打印。接下来，未等该字符打印完，就去调用系统的调度器，选择另一个就绪任务去运行，而当前的这个任务，就会被阻塞起来。

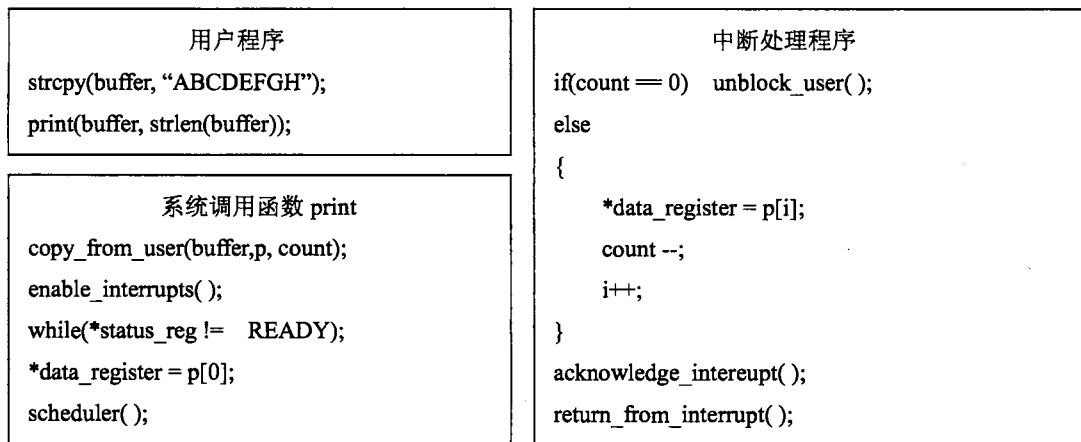


图 3-62 中断驱动的例子

当打印机完成一个字符后，将向 CPU 发出一个中断。在中断处理程序当中，首先判断一下，如果所有的字符都已打印完，那么就阻塞队列中，把用户任务唤醒，使它处于就绪状态；如果还有字符需要打印，就直接把下一个字符复制到打印机的数据寄存器当中，启动打印操作，而不需要再去循环地判断打印机是否就绪。接下来是一些后继处理，先向中断控制器发出一个确认信号，然后结束中断处理程序，返回到被中断的那个任务。

### 3. 直接内存访问方式

在中断驱动的控制方式下，每一次数据读写还是通过 CPU 来完成，而且每一次处理的数据量很少，如 1 个字节，所以中断出现的频率就很高。而中断处理需要额外的系统开销，所以也会浪费一些 CPU 时间。因此人们又提出了一种新的解决办法，也就是直接内存访问 (Direct Memory Access, DMA) 的控制方式。它的基本思路是：让 DMA 控制器来代替 CPU，完成 I/O 设备与内存之间的数据传送，从而空出更多的 CPU 时间，去运行其他的任务。

仍以打印字符的问题为例。如图 3-63 所示，在 DMA 控制方式下，用户程序所做的事情是完全相同的，即把字符串复制到一个缓冲区 buffer 当中，然后调用系统函数 print。在 print 函数当中，首先也是把 buffer 当中的字符串复制到系统内核的缓冲区 p 当中，然后对 DMA 控制器进行编程，设置它的各个寄存器的内容，包括内存起始地址、需要打印的字符个数、数据传输的方向等。之后，print 函数就完成了任务，所以就调用系统的调度程序，



选择另一个就绪任务去运行，而当前的这个任务就会被阻塞起来。接下来，当 CPU 正在执行这个新任务的同时，DMA 控制器会与设备控制器进行交互，把需要打印的字符，一个接一个地送到打印机控制器当中。在所有的字符都打印完之后，就向 CPU 发出一个中断，表明这一次的 I/O 操作已经全部完成了。因此，在中断处理程序里面，已经没有什么实质性的工作，先是向中断控制器发出一个确认信号，然后唤醒刚才被阻塞的任务。

系统调用函数 print	中断处理程序
<pre>copy_from_user(buffer,p, count); set_up_DMA_controller(); scheduler();</pre>	<pre>acknowledge_intereupt(); unblock_user(); return_from_interrupt();</pre>

图 3-63 直接内存访问的例子

采用 DMA 控制方式，最大的优点是减少了中断的次数。原本每打印一个字符，都要产生一次中断，而现在当所有的字符都打印完后，才会产生一个中断，这样就减少了中断处理的开销。

### 3.5.3 I/O 软件

设备管理软件的设计水平决定了设备管理的效率。为了更好地管理系统当中各式各样的 I/O 设备，在软件上通常采用分层的体系结构，把各种设备管理软件组织成一系列的层次。其中，低层软件是面向硬件的，与硬件特性密切相关，它把硬件同上层的软件隔离开来。而较高层的软件是面向用户的，负责向用户提供一个友好、清晰、统一的编程接口。一般来说，这个层次结构可以分四层：中断处理程序、设备驱动程序、设备独立的 I/O 软件 and 用户空间的 I/O 软件。

#### 1. 中断处理程序

中断处理程序与设备驱动程序密切配合，来完成特定的 I/O 操作。当一个用户程序需要某种 I/O 服务时，它会去调用相应的系统函数，而这个函数又会去调用相应的设备驱动程序。然后，在驱动程序中会启动 I/O 操作，并且被阻塞起来，直到这个 I/O 操作完成后，将产生一个中断，并跳转到相应的中断处理程序。之后在中断处理程序中，将会唤醒被阻塞的驱动程序。至于阻塞和唤醒的具体实现，可以采用各种任务间通信的方式，如 P、V 原语。

在中断处理过程中，需要执行不少指令，例如，保存 CPU 的运行上下文、为中断服务子程序设置一个运行环境、向中断控制器发出应答信号、执行相应的中断服务子程序等。这些都需要一定的时间开销。

## 2. 设备驱动程序

设备驱动程序是直接同 I/O 设备打交道，直接对它们进行控制的软件模块。设备驱动程序的基本任务就是接收来自于上层 I/O 软件的抽象请求，并且去执行这个请求，例如，抽象的读操作、写操作、设备的初始化操作等。上层的 I/O 软件通过这些抽象的函数接口与设备驱动程序打交道，这些接口是标准的、稳定不变的，而硬件设备的具体细节被封装在设备驱动程序里面。这样，如果硬件设备发生了变化，只要更新相应的设备驱动程序即可，不会影响到上层软件对它的使用。

设备驱动程序与具体的设备类型密切相关。每一个 I/O 设备都需要相应的设备驱动程序，而每一个设备驱动程序一般也只能处理一种类型的设备，因为对于不同类型的设备，它们的控制方式是不同的。例如，对于一个鼠标驱动程序来说，它需要从设备控制器中读取各种各样的信息，包括鼠标移动的位置、哪一个键被按下了等；而对于一个磁盘驱动程序来说，它为了进行磁盘的读写操作，就必须知道扇区、磁道、柱面、磁头等各种各样的参数，并使用这些参数来控制磁盘控制器。

一般而言，在具体实现一个设备驱动程序的时候，可以采用一种通用的结构。

(1) 检查输入的参数是否有效，如果无效，就返回一个出错报告；如果有效，就把输入的抽象参数转换为控制设备所需要的具体参数。

(2) 检查设备当前是否空闲，如果设备正忙，那么这一次的 I/O 请求就暂时没法完成，所以把它加入等待队列，稍后再处理；如果设备空闲，再检查硬件的状态，看是否具备了运行的条件。

(3) 设备驱动程序向设备控制器发出一连串的命令，也就是说，把这些命令写入到控制器的各个寄存器当中。

(4) 在发出控制命令后，如果这个 I/O 操作需要一定的处理时间，不能马上完成，那么驱动程序就会把自己阻塞起来，直到 I/O 操作完成。这时会发生一个中断，在这个中断处理程序里面把驱动程序唤醒。

(5) I/O 操作完成后，驱动程序还要检查出错情况。若一切正常，就返回一些状态信息给调用者。如果这是一个输入操作，还要把输入的数据上传给上一层的系统软件。

在实时内核的 I/O 系统中，用户 I/O 请求在到达设备驱动程序之前，通常都只进行非常少量的处理。事实上，实时内核的 I/O 系统的作用就像一个转换表，把用户对 I/O 的请求转换到相应的设备驱动程序中。这样，驱动程序就能获得最原始的用户请求，并对设备进行操作。

## 3. 设备独立的 I/O 软件

在设备驱动程序的上面，是设备独立的 I/O 软件。它是系统内核的一部分，主要任务是实现所有设备都需要的一些通用的 I/O 功能，并向用户级的软件提供一个统一的访问接



口。具体来说,在这个层面上实现的功能主要有:设备驱动程序的管理、与设备驱动程序的统一接口、设备命名、设备保护、缓冲技术、出错报告以及独占设备的分配和释放。

设备驱动程序的管理通过驱动程序地址表来实现。驱动程序地址表中存放了各个设备驱动程序的入口地址,可以通过此表来实现设备驱动的动态安装与卸载。

操作系统的一个主要问题就是如何使各种 I/O 设备与设备驱动程序的处理方式大致相同,从而方便系统的设计和用户的使用,实现设备独立性。因此, I/O 系统通常会提供一个统一的调用接口,包含了一些常用的设备操作,如设备初始化、打开设备、关闭设备、读操作、写操作、设备控制等。在 I/O 设备的命名规则上,可以采用统一命名的方式,然后由设备独立的 I/O 软件来负责把设备的符号名映射到相应的设备驱动程序。

缓冲技术是操作系统当中很重要的一种技术,它的基本思想是:在实现数据的 I/O 操作时,为了缓解 CPU 与外部设备之间速度不匹配的矛盾,提高资源的利用率,可以在内存当中开辟一个空间,作为缓冲区,当需要从设备读取数据时,先到缓冲区中去查找,如果能够找到,就不用去访问外设了。同样,在往设备中写入数据时,也是先写到缓冲区中。这样,如果马上又要用到这些数据,就可以直接从缓冲区中去取。缓冲技术是一种实用、有效的技术,因为对于 I/O 设备的访问,也会满足程序的局部性原理,即在访问设备数据的时候,在一小段时间内,可能会集中地访问其中的若干个数据块。因此设置缓冲区可以减少对 I/O 设备的访问,从而提高系统的性能。在具体实现上,缓冲技术可以分为单缓冲、双缓冲、多缓冲和环形缓冲。

#### 4. 用户空间的 I/O 软件

通常大部分的 I/O 软件都是包含在操作系统当中的,是操作系统的一部分,但也有一小部分的 I/O 软件,它们运行在系统内核之外。这主要可以分为以下两种。

- 与用户程序进行链接的库函数。例如,在 C 语言中与 I/O 有关的各种库函数。不过,对于这些库函数,它们在具体实现的时候,其实是把传给它们的参数再往下传递给相应的系统函数,然后由后者来完成实际的 I/O 操作。
- 完全运行在用户空间当中的程序。例如, Spooling 技术是在多道系统中,一种处理独占设备的方法。

Spooling (simultaneous peripheral operations on line) 是“外围设备联机操作”的缩写,常称为 spooling 技术。假脱机技术或虚拟设备技术。它可以把一个独占的设备转变为具有共享特征的虚拟设备,从而提高设备的利用率。它的基本思想是:在多道系统当中,对于一个独占的设备,专门利用一道程序,即 Spooling 程序,来增强该设备的 I/O 功能。具体来说,一方面, Spooling 程序负责与这个独占的 I/O 设备进行数据交换,这可以称为实际的 I/O 操作。另一方面,应用程序在进行 I/O 操作时,只是同这个 Spooling 程序交换数据,这可以称为虚拟的 I/O 操作。此时,它实际上是与 Spooling 程序当中的缓冲区打交道,从

中读出数据或往里写入数据，而不是直接同实际的设备进行 I/O 操作。

Spooling 技术的优点有两个。第一，它能提供高速的虚拟 I/O 服务。应用程序的虚拟 I/O 比实际的 I/O 速度要快，因为它只是在两个任务之前的一种通信，把数据从一个任务交给另一个任务，这种交换是在内存中进行的，而不是真正地让机械的物理设备去运作，这样就能缩短应用程序的执行时间。第二，它能实现对独占设备的共享，也就是说，由 Spooling 程序提供虚拟设备，然后各个用户任务就可以对这个设备依次地共享使用。

## 3.6 文件系统

### 3.6.1 嵌入式文件系统概述

所谓的文件系统，就是操作系统中用以组织、存储、命名、使用和保护文件的一套管理机制。嵌入式文件系统就是应用在嵌入式系统中的文件系统，它是嵌入式系统的一个重要组成部分。随着嵌入式硬件设备的广泛应用、价格的不断降低以及嵌入式应用范围的不断扩大，嵌入式文件系统的重要性显得更加突出。

由于应用背景和系统结构的不同，嵌入式文件系统与桌面文件系统在很多方面有较大的区别。例如，在普通桌面操作系统中，文件系统不仅要管理文件，提供文件系统的 API 接口函数，还要管理各种设备，支持对设备和文件操作的一致性。而在嵌入式文件系统中，情况则有所不同。在某些情况下，嵌入式操作系统可以针对特殊的目的进行定制，这就对嵌入式操作系统的功能完整性和可伸缩性提出了更高的要求。一般来说，嵌入式文件系统要为嵌入式系统的设计目的服务，对于不同用途的嵌入式操作系统，它们的文件系统在许多方面也各不相同。

在嵌入式系统中，文件系统存在于不同类型的存储设备当中，如 Flash、RAM 和硬盘。它通常是以中间件或应用程序的形式安装（mount）在存储设备上。常见的一些嵌入式文件系统有下面几种。

- **FAT (File Allocation Table):** FAT 文件系统是最常用的文件系统之一，最早于 1982 年应用在 MS-DOS 操作系统当中。许多嵌入式操作系统都支持 FAT 文件系统，如 VxWorks、QNX、Windows CE 等。为了与 PC 机文件系统兼容，在嵌入式系统设计中一般使用标准的 FAT12/16/32 文件系统。
- **NFS (Network File System):** 网络文件系统，基于远程过程调用 (Remote Procedure Call, RPC) 和扩展数据表示 (Extended Data Representation, XDR)。它可以将外部设备安装在文件系统中，就好像是一个本地的文件分区，从而可以实现对网络文件的快速、无缝共享。

- FFS (Flash File System): 用于 Flash 存储器的文件系统。
- DosFS: 用于实时条件下的块设备 (磁盘) 访问, 并且与 MS-DOS 文件系统兼容。
- RawFS: 提供了一个简单的“生”的文件系统, 它的基本思路是把整个磁盘视为一个巨大的文件。
- TapeFS: 用于磁带设备, 在磁带上不使用标准的文件或目录结构。其基本思路是: 把整个磁带卷视为一个巨大的文件。
- CdromFS: ISO 9660 标准文件系统, 用于 CD-ROM 数据的访问。

### 3.6.2 文件和目录

#### 1. 文件的基本概念

从用户的角度来说, 文件是一种抽象的机制, 它提供了一种把信息保存在磁盘等外部存储设备上, 并且便于以后访问的方法。这种抽象性体现在用户不必去关心具体的实现细节, 例如, 这些信息被存放在什么地方、是如何存放的等。

当一个文件被创建时, 必须给它指定一个名字, 因为用户就是通过文件名来访问这个文件的。文件名是一个有限长度的字符串, 它一般由两个部分组成: 文件名和扩展名。对于有的系统来说, 文件名的长度一般不超过八个字符, 但是很多系统现在已经支持长文件名。

文件的逻辑结构指的是文件系统向外提供给用户的文件结构形式, 它独立于文件在磁盘上的物理存储结构。文件的逻辑结构主要有三种: 无结构、简单的记录结构和复杂结构。对于现代文件系统, 通常采用的是无结构的形式。也就是说, 整个文件是由一系列无结构的字节流所组成的, 文件的大小也就是这些字节的个数。如图 3-64 所示, 中间的横线表示一个用户接口, 在它的下面是文件系统, 上面是用户程序。对于文件系统来说, 所谓的文件就是由很多个字节所组成的字节流, 至于每个字节之间有什么样的关系, 有什么样的结构, 它并不知道。当然, 在用户程序的内部, 在具体使用该文件时, 它的确是有结构的,

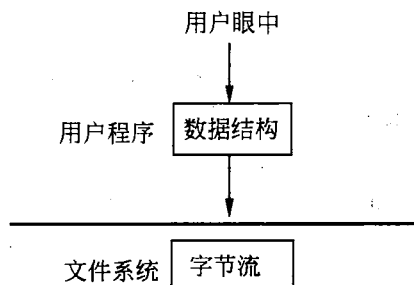


图 3-64 文件的逻辑结构

如数组结构、记录结构、树型结构等，这完全是由用户程序自己来设计和维护的，与文件系统无关。这样使用起来就非常的方便和灵活。

文件可以按照不同的准则来进行分类，例如：

- 按照文件的性质和用途，可将文件分为系统文件、库文件和用户文件。
- 按照文件的保护方式，可将文件分为只读文件、读写文件和可执行文件。
- 按照文件的功能，可将文件分为普通文件和目录文件。普通文件就是通常意义上所说的文件，它里面包含了用户的各种数据；目录文件是一种专用的系统文件，用来管理文件系统的组织结构。
- 在有些操作系统中，还有一种特殊的设备文件，也就是说，用文件的形式来管理 I/O 设备，包括字符设备文件和块设备文件，前者用来描述串行的 I/O 设备，后者用来描述磁盘等块设备。

除了文件名之外，操作系统还会给每一个文件附加一些其他的信息，这些信息称为文件的属性。对于不同的操作系统，文件属性的类型和个数各不相同，一般来说，都会包含以下一些属性：文件的保护信息、文件的创建者、只读标志位、隐藏标志位、系统标志位、文件的创建时间、最近访问时间、最近修改时间以及文件的长度信息等。

## 2. 文件的使用

文件的使用讨论的是操作系统所提供的与文件有关的系统调用。

### (1) 文件的存取方法

文件的存取方法可以分为两类。

- 顺序存取：对于文件中的每一个字节或记录，只能从起始位置开始，一个接一个地顺序访问，不能跳跃式访问。这是早期的操作系统所提供的存取方式。
- 随机存取：根据所需访问的字节或记录在文件中的位置，将文件的读写指针直接移至该位置，然后进行存取，其中每一次存取操作都要指定该操作的起始位置。现代操作系统都提供随机存取的方式。

### (2) 文件的访问

文件的访问指的是与文件内容读写有关的各种文件操作，包括以下几种。

- 打开操作 (open)：在访问一个文件前，必须先打开它。
- 关闭操作 (close)：在使用完一个文件后，要关闭该文件。
- 读操作 (read)：从文件中读取数据。
- 写操作 (write)：把数据写入文件。
- 添加操作 (append)：把数据添加到文件的末尾。
- 定位操作 (seek)：指定文件访问的当前位置。

### (3) 文件的控制

文件的控制指的是与文件属性控制有关的各种文件操作，包括文件的创建、删除、读取文件属性、设置文件属性、修改文件名等。

### 3. 目录

为了对系统中的文件进行组织和管理，人们引入了目录（directory）的概念。目录也称文件夹（folder），它是一张表格，记录了在该目录下每个文件的文件名和其他的一些管理信息。一般来说，每个文件都会占用这张表格的某一行，即一个目录项。由于文件系统中的目录是动态创建的，其大小是不断变化的，因此，目录通常都是以文件的形式存放在磁盘上。另外，在目录的管理上，也有一些相关的系统调用，如创建目录、删除目录、修改目录名等。

为了更好地组织文件，提高文件的访问效率，在目录的逻辑结构上，通常采用的是多级目录结构，也称树状目录结构或层次目录结构，其形状好像是一棵倒立的树，树的根节点称为根目录。在每一个目录下，既可以增加普通的文件，也可以增加新的子目录。

在多级目录结构中，如何来指定需要访问的文件或目录呢？主要有两种方法。

- 绝对路径名：对于每一个文件或目录，可以用从根目录开始依次经由的各级目录名，再加上最终的文件名或目录名来表示，在每一级目录名之间，用分隔符隔开。一个文件或目录的绝对路径名是唯一的，例如\spell\mail\copy\all。
- 相对路径名：用户首先指定一个目录作为当前的工作目录（working directory），然后在访问一个文件或目录时，可以使用相对于当前工作目录的部分路径名，即相对路径名。例如，假设当前的工作目录是\spell\mail\copy，那么使用相对路径名 all 的效果等价于使用绝对路径名\spell\mail\copy\all。

### 3.6.3 文件系统的实现

文件系统的实现讨论的是文件和目录是如何来实现的，磁盘空间是如何来管理的，如何才能使整个文件系统高效、可靠地运转。

#### 1. 数据块

如前所述，文件的逻辑结构一般是字节流，即无结构。用户程序可以在这种字节流的基础上，构造自己所需的各种数据结构。文件是存放在磁盘等存储设备当中的，而这些设备的访问单元并不是字节。例如，在磁盘中，是以扇区为单元来进行读写操作的。因此，对于文件系统而言，必须将用户提交的这种字节流（一个连续的逻辑地址空间）映射为磁盘所需要的扇区。为了实现设备的独立性，通常的做法是把磁盘空间划分为一个个大小相同的块（block），称为物理块，每个物理块包含若干个连续的扇区，同时把文件的字节流也分成大小相同的逻辑块，然后在文件系统的内部，以块为单位来进行操作，把每一个逻辑块保存在一个物理块当中。

## 2. 文件的实现

文件的实现需要解决两个方面的问题：一是如何来描述一个文件，用什么样的数据结构来记录文件的各种管理信息；二是如何来存储文件，如何把文件的各个连续的逻辑块存放到磁盘的空闲物理块当中，并记录逻辑块与物理块之间的映射关系。

### (1) 文件控制块

文件的描述方法就是文件控制块 (File Control Block, FCB)，它是操作系统为了管理文件而设置的一种数据结构，里面存放了与一个文件有关的所有管理信息。FCB 是文件存在的标志。

对于不同的操作系统，它们的文件控制块所包含的内容是各不相同的。一般来说，主要包含两类信息。

- 文件的属性信息：包括文件的类型和长度、文件的所有者、文件的访问权限、文件的创建时间、最后访问时间以及最后修改时间等。
- 文件的存储信息：文件在磁盘上的存放位置，它被存放在哪一些物理块当中。

### (2) 文件的物理结构

文件的物理结构研究的是如何把一个文件存放在磁盘等物理介质上。具体来说，就是以块为单位，研究如何把文件的一个个连续的逻辑块存放在不同的物理块当中，从而建立逻辑块与物理块之间的映射关系。文件的物理结构主要有三种形式：连续结构、链表结构和索引结构。

#### ① 连续结构

连续结构也叫顺序结构，它的基本思想是把文件的各个逻辑块按照顺序存放在若干个连续的物理块当中。这种方法的优点是简单、易于实现。对于一个文件，系统只要记住它的第一个物理块的编号和物理块的个数，就可以通过简单的加法来实现从逻辑块到物理块的映射。例如，假设一个文件总共有  $N$  个逻辑块，而且第一个逻辑块是存放在第  $X$  个物理块当中，那么可以推算出第  $i$  个逻辑块是保存在第  $X+i-1$  个物理块当中。

连续结构也有它的缺点：首先，随着磁盘文件的增加和删除，将会形成空闲物理块与占用物理块相互交错的情形，这样，那些比较小的物理块区域就无法再利用，成为外碎片；其次，在连续结构方式下，文件的大小不能动态地增长。

连续结构主要用在 CD-ROM、DVD 等一次性写入的光学存储介质当中。对于这些应用，文件的大小都是已知的，所以是固定不变的，所以连续结构的两个缺点都不存在了，而优点还保留着。

#### ② 链表结构

链表结构的基本思路是：把文件的各个逻辑块依次存放在若干个物理块当中，这些物理块既可以是连续的，也可以是不连续的，然后在各个块之间通过指针连接起来，前一个

物理块指向下一个物理块，从而形成一条链表。在具体实现链表结构时，需要在每一个物理块当中，专门利用若干字节来作为指针，指向下一个物理块。对于一个文件，系统只要记住它的链表的首节点指针，就可以定位到这个文件中的任何一个物理块。

链表结构克服了连续结构的缺点。由于不连续的物理块之间可以通过指针连接起来，所以每一个物理块都能够用上，不存在外碎片的问题，而且文件的大小也可以动态变化。

链表结构的缺点是：在访问一个文件时，只能顺序地进行访问，如果想随机地访问，那么速度会比较慢。例如，为了访问一个文件的第  $n$  个逻辑块，文件系统必须从这个文件的第一个物理块开始，按照每一个物理块当中的链表指针，顺序地去遍历前  $n$  个块，所以时间比较长。

为了解决这个问题，人们又对链表结构进行了改进，提出了带有文件分配表的链表结构。它的基本思路是：在链表结构的基础上，把每一个物理块当中的链表指针抽取出来，单独组成一个表格，也就是文件分配表（File Allocation Table, FAT），并把它存放在内存当中，然后，如果要随机地去访问文件的第  $n$  个逻辑块，可以先从 FAT 表中查到相应的物理块地址，之后根据这个地址直接去访问磁盘，这样速度就较快。

文件分配表的具体实现是，在整个文件系统中设置一个一维的线性表格，它的表项个数就等于磁盘上物理块的个数，并按照物理块编号的顺序来建立索引。对于系统中的每一个文件，在它的文件控制块中记录了这个文件的第一个物理块的编号  $X_1$ ，然后在 FAT 表的第  $X_1$  项中，记录了该文件的第二个物理块编号  $X_2$ 。就这样一直下去，从而形成了一个链表。在链表的最后一个节点中，存放了一个特殊的文件结束的标识。

图 3-65 所示是文件分配表的一个例子。通过文件 1 的目录项可以知道，它的第一个逻辑块存放在第一个物理块中。然后去查询 FAT 表，可以知道，它的第二、第三个逻辑块分别存放在第二、第三个物理块中。FAT 表的第三项是一个特殊的值  $0xFFFF$ ，表明文件的结束，所以该文件总共有三个块。类似的，文件 2 也有三个数据块，分别存放在第四、第五和第七个物理块中。

### ③ 索引结构

索引结构的基本思路是：把文件当中每一个逻辑块所对应的物理块编号直接记录在这个文件的文件控制块当中，这样的文件控制块称为是 I 节点，或索引节点（index node）。这样，对于系统中的每一个文件，都有一个自己的索引节点。通过这个索引节点，就能够直接实现逻辑块与物理块之间的映射关系。例如，如果要去访问文件的第  $i$  个逻辑块，可以先到其索引节点的地址映射表中查一下第  $i$  项的内容，就可以知道相应的物理块编号，然后就可以直接去访问磁盘了。

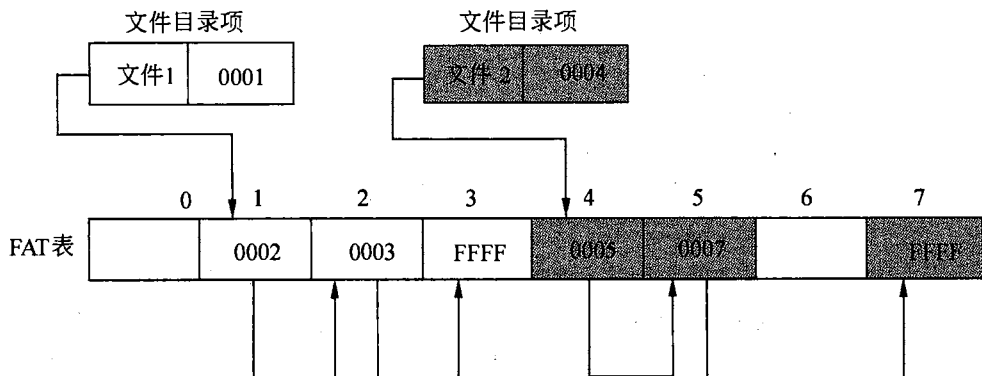


图 3-65 FAT 文件分配表的例子

### 3. 目录的实现

在实现目录时，不同的文件系统采用了不同的实现方法。一般来说，可以分为两种类型：即直接法和间接法。

- **直接法:**如图 3-66 (a) 所示, 它把文件控制块的内容直接保存在目录项当中。因此, 每一个目录项就等于某个文件的文件名加上它的 FCB, 包括文件的各种属性信息和它在磁盘上的存放位置。

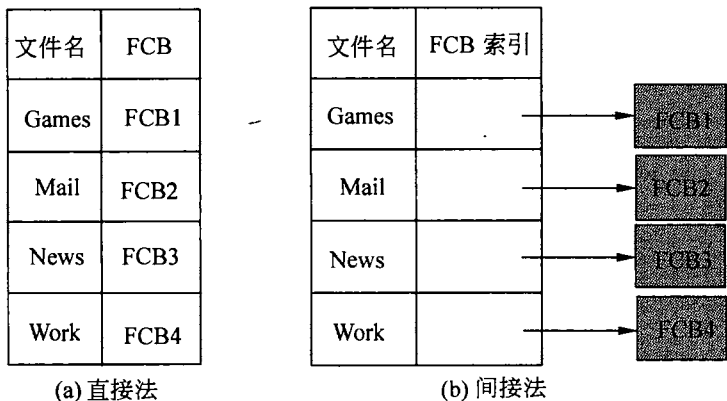


图 3-66 目录的实现方法

- 间接法：如图 3-66（b）所示，每一个文件的 FCB 不是保存在它的目录项当中，而是单独存放。然后在每一个目录项里面，只有两个内容：文件名和该文件的 FCB 所在的地址。



不管是哪一种类型的实现方法，目录的基本功能都是一样的，即如果用户给出一个文件名，就返回相应文件的 FCB。

#### 4. 空闲空间管理

为了管理磁盘上的空闲空间，系统会维护一个空闲空间列表，上面记录了磁盘上所有的空闲物理块。在具体实现这个空闲列表时，主要有三种方法：位图法、链表法和索引法。

- 位图法：每一个物理块用一个位（bit）来表示。如果该物理块空闲，相应位的值为 1；如果该物理块已分配，相应位的值为 0。若磁盘有  $N$  个物理块，则对应于  $N$  个 bit。然后将这些连续的位流分隔为一个个字节，每八位一个字节，再把这些字节组织成一个个字，如每个字四个字节，这样就得到了相应的位图。
- 链表法：在每一个空闲的物理块上都有一个指针，然后把所有的空闲块通过这个指针连接起来，形成一个链表。文件系统只要记住这个链表的首节点指针，就可以去访问所有的空闲物理块。
- 索引法：对链表法的一种修改。同样构造一个空闲链表，但是这个链表中的物理块本身并不参与分配，而是专门用来记录系统中其他空闲物理块的编号（索引）。

## 第 4 章 嵌入式软件程序设计

### 4.1 嵌入式软件开发概述

#### 4.1.1 嵌入式应用开发过程

一个嵌入式应用项目的开发过程是一个硬件设计和软件设计的综合过程，也是一个系统过程。一般而言，要经历以下几个步骤：

(1) 硬件的设计与实现，包括元器件选型、原理图编制、印制板设计、样板试制、硬件功能测试等。

(2) 设备驱动软件的设计与实现，包括引导加载程序（BootLoader）的编写、各种设备驱动程序的编写。

(3) 嵌入式操作系统的选择、移植，以及 API 接口函数的设计。

(4) 支撑软件的设计与调试。

(5) 应用程序的设计与调试。

(6) 系统联调，样机交付。

由此可见，开发一个嵌入式应用其实就是开发一个特定用途的计算机系统，开发时需要综合考虑系统软硬件各个层次上的所有问题。因此，无论是开发的过程还是开发的环境，都与一般的桌面系统上的应用程序开发有着显著的不同，需要考虑更多的因素。仅软件部分就要考虑板级支持包 BSP 的开发、操作系统的移植、应用程序的开发和操作系统的接口等问题。即使只开发应用程序，也要在工程项目中将操作系统文件、设备驱动文件和应用程序文件集成在一起，经过修改整理再编译成目标文件。

图 4-1 所示是一个典型的嵌入式应用程序的生成和加载过程。

#### 4.1.2 嵌入式软件开发的特点

嵌入式软件开发有如下的几个特点。

(1) 需要交叉编译工具

嵌入式系统目标机上的资源非常有限，例如，处理器的结构比较简单，速度较慢；内存和外存的容量小；显示功能较弱；软件资源较少，等等。因此，直接在嵌入式系统的硬

件平台上开发和调试应用软件比较困难,有时甚至是不可能的。目前一般采用的解决办法是:把集成开发环境安装在高性能的 PC 机上,然后在 PC 机上进行嵌入式应用软件的开发。由于 PC 机上的 CPU 芯片一般是 x86 芯片,而嵌入式系统中的处理器芯片种类繁多,如 ARM、MIPS、PowerPC 等,两种处理器的指令集是不同的。因此,在集成开发环境下编写的源程序需要经过交叉编译,才能生成可在目标平台上运行的二进制代码格式。

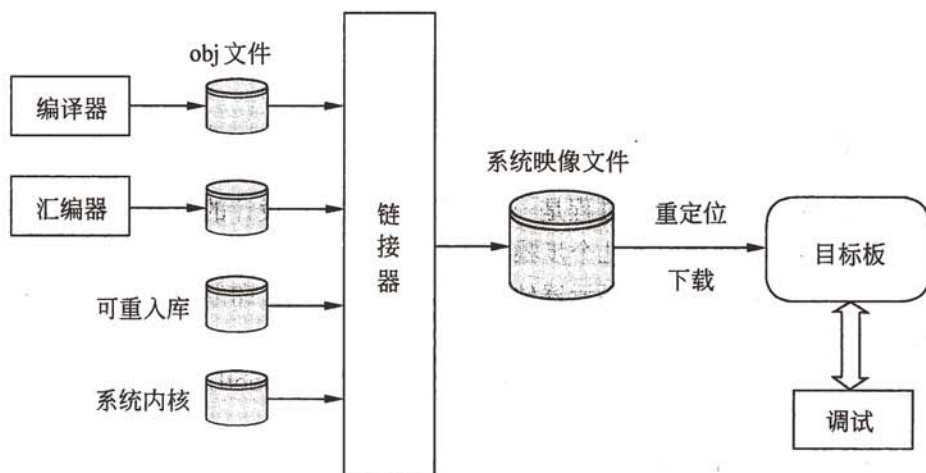


图 4-1 嵌入式应用程序的生成与加载

## (2) 通过仿真手段进行调试

目标机执行程序经过交叉编译之后,还要经过调试排错,确认能正常运行后才能交付使用。显然,在目标机上进行调试排错是非常困难的,所以实际的做法是仿真调试。也就是通过接口和信号线,把目标机上机器指令的执行结果和 CPU 当前各个寄存器的值传送到集成开发平台,使开发人员能够观察到目标机的执行状况,从而判断出指令执行的正确与错误。

## (3) 开发板是中间目标机

嵌入式应用软件需要在开发板上完成所有的开发任务。开发完成之后,才把目标程序安装在目标机上运行。也就是说,开发板只是中间的目标机,它的任务就是开发和调试。

## (4) 可利用的资源有限

在台式机环境下,程序员拥有大量的硬件和软件编程资源,对内存容量、硬盘容量、可以打开的文件数量等问题几乎不用操心。然而在嵌入式软件开发中,就必须考虑可用资源的问题。嵌入式系统使用的处理器、内存和存储容量与台式机相比有很大的差距。因此,嵌入式代码不仅要提供丰富的功能,还必须满足其他的一些约束条件,如按所需速度运行

以满足系统期限、适应内存总量限制、满足功耗要求等。

#### (5) 需要与硬件打交道

在开发桌面应用程序时，很少直接与硬件打交道，除非是开发初级的设备驱动程序。但是在嵌入式系统的开发中，软件与硬件的关系非常密切，经常需要对运算器、寄存器和存储器进行操作，即使是采用了嵌入式操作系统，为了结构的简化以及节省空间，也要容许应用程序直接去访问外围的寄存器，特别是当系统发生了无法理解的错误时，可能无法确认到底是程序写错，还是目标平台的线路有问题。因此程序员除了要了解如何编写高级语言程序（C、C++、Java）与初级语言程序（汇编）外，还要了解硬件设计及除错的知识。

### 4.1.3 嵌入式软件开发的挑战

在开发嵌入式软件时，会面临如下一些挑战和问题。

#### (1) 软硬件协同设计

嵌入式系统由硬件和软件组成，因此，在系统设计时，需要考虑哪一些功能用硬件来实现，哪一些功能用软件来实现。硬件实现的优点是速度快，缺点是芯片成本高，耗电量大，且需要占用额外的空间；软件实现的优点是灵活性，如果算法发生了改变，那么修改软件是很容易的。例如，TCP/IP 协议栈的实现，几十年来，都是用软件来实现，因为这种方法为改变协议提供了灵活性。在台式机环境下，TCP/IP 协议栈被绑定在操作系统中，这是可以接受的，因为桌面计算机有大量的内存和外存容量。不过，现在已经出现了 TCP/IP 协议栈的单芯片实现方案，这种方法极大地加快了协议的处理过程。它的另一个优点就是可以把它集成到嵌入式硬件中，从而使嵌入式系统具备网络功能。

#### (2) 嵌入式操作系统

嵌入式应用的开发可以分为两种情况。

- 无操作系统：嵌入式软件的设计主要是以应用为核心，应用软件直接建立在硬件上，没有专门的操作系统，软件的规模也很小，基本上属于硬件的附属品。开发人员可以混合使用汇编语言和 C 语言，实现存储管理、I/O 管理和任务管理等服务。这种方式的优点是：软件是为特定的应用而专门编写的，因而代码的结构紧凑，容量小、效率高，既提高了运行速度，又节省了存储空间。
- 有操作系统：首先将一个可用的嵌入式操作系统移植到目标处理器，然后当程序员在开发应用程序时，不是直接面对嵌入式硬件设备，而是在操作系统的基础上编写，操作系统会提供必要的 API 函数来实现各种功能。在这种情况下，开发人员不必操心存储管理、任务管理等一般性的事务，而是将精力集中在应用软件的开发上，因而开发速度更快，编写出来的代码更加可靠。这也是现在广泛采用的一种开发方法。

### (3) 代码优化

一般来说,桌面应用软件的开发人员不必过多地去考虑代码的优化,因为处理器的功能强大,内存的容量也足够用,而且在响应时间上,即使有几秒钟的差异也不会带来显著的影响。但是在嵌入式系统中,存储器容量和执行时间通常是最主要的约束条件——尽管编译器会实现代码上的优化,但编程人员仍必须精心编写代码,并对代码进行优化,开发出运行速度快、存储空间少、维护成本低的软件。有时,为了达到系统所要求的响应时间,编程人员可能需要使用汇编语言来编写部分代码。

### (4) 有限的 I/O 功能

在台式机环境下,一般都使用键盘和鼠标作为输入设备,显示器作为输出设备。但是在嵌入式系统中,不一定存在这些外设。事实上,大多数嵌入式系统的 I/O 功能是有限的。例如,只有小键盘(具有 8 个或 12 个功能键)可用于输入数据;在输出设备上,可能只有少量的 LED,或每行 8 到 12 个字符且仅有两行的小型 LCD 显示器。而有些嵌入式系统根本就没有键盘或显示器这样的 I/O 设备来与用户进行交互。例如,在许多过程控制系统中,采用电信号来作为输入并产生电信号来作为输出。因此,开发、测试和调试这一类的系统更具有挑战性,必须采用特殊的程序来测试这些系统。

总之,嵌入式软件的开发具有很大的挑战性,需要开发人员具有扎实的软、硬件基础,能灵活运用不同的开发手段和工具,具有较丰富的开发经验。如果要设计出可靠、稳定、高效的嵌入式软件,就必须综合考虑多种因素,如并行性、兼容性、实时性、层次性、可扩展性、有限的资源、多样性和可读性等。

## 4.2 嵌入式程序设计语言

程序设计语言是为了编写计算机程序而人为设计的符号语言,用于对计算机过程进行描述、组织和推导。1957 年出现的 FORTRAN 是第一个被广泛应用的高级语言,经过几十年的发展,目前世界上流行的程序设计语言有上百种之多,程序设计语言的演化速度已经超越了运行它们的机器。

与通用软件相比,嵌入式软件具有自身的一些特点,如规模较小、实时性和可靠性要求高、与硬件结合紧密等。因此,并不是所有的编程语言都适合于嵌入式软件的开发,开发人员必须根据具体的应用需求和软硬件条件,来选择合适的编程语言。

### 4.2.1 程序设计语言概述

#### 1. 低级语言与高级语言

机器语言是与计算机硬件关系最为密切的一种计算机语言,在计算机硬件上执行的就

是一条条用机器语言来编写的指令。对于任何一个计算机程序来说，都要先把它变成机器指令的形式，然后才能够在计算机上运行。不过，这种指令完全是由二进制的0和1所组成，一条指令就是由若干个0和若干个1所组成的一个字符串，所以很难看懂。例如，图4-2所示就是一段机器语言代码，每一行表示一条指令，有的指令长一点，有的指令短一点。

```
100010110100010111111100
00111011010001011111000
0000100001111110
100010110100110111111100
100010010100110111110100
1110101100000110
10001011010101011111000
100010010101010111110100
```

图 4-2 一段机器语言代码

显然，对于那些不太熟悉机器语言的人来说，看到这些0、1所组成的字符串，完全就像看天书一样，根本就看不懂。所以说，如果要采用机器语言来编写程序的话，那么工作效率将会极其低下，而且编写出来的代码，它的正确性也很难保证。例如，在图4-2所示的某条指令当中，如果不小心把其中的一个1写成了0，那么这条指令的含义可能就完全变了，而整个程序的运行结果可能就完全不同了。

为了提高程序设计的效率，在20世纪50年代，人们提出了汇编语言的概念。它的基本思路是：用符号的形式来代替二进制的指令。例如，对于一条机器指令00110101，我们可能不知道它的作用是什么，但如果用符号ADD来代替它，那么这条指令的功能就很容易猜到，它是一个加法运算。所以说，采用汇编语言来编写程序，就比机器语言要方便得多，也容易得多。不过，在使用汇编语言后，虽然编程的效率和程序的可读性都有所提高，但汇编语言同机器语言非常接近，它的书写格式在很大程度上取决于特定计算机的机器指令，所以它仍然是一种面向机器的语言。人们通常把机器语言和汇编语言统称为低级语言。

为了更好地进行程序设计，提高程序设计的效率，人们又提出了高级程序设计语言的概念。它的基本思路是：用一种更自然、更接近于人类语言习惯的符号形式（如数学公式）来编写程序，这样写出来的程序更容易理解和使用。目前已经开发出许多高级语言，如FORTRAN、COBOL、PASCAL、C、Ada、C++和Java等。

## 2. 汇编程序、编译程序和解释程序

尽管人们可以借助汇编语言和高级语言与计算机进行交互，但是计算机仍然只能理解

和执行由 0、1 序列构成的机器语言程序，所以我们用汇编语言或高级语言编写出来的程序并不能直接地在计算机上运行，而需要先经过一个专门的翻译，把它变成机器指令的形式，然后才能够运行。担任翻译任务的程序称为语言处理程序。由于应用的不同，对语言的翻译也是多种多样的，它们大致可以分为三种类型：汇编程序、解释程序和编译程序。

用汇编语言或某种高级语言编写的程序称为源程序，源程序不能直接在计算机上执行。如果源程序是用汇编语言编写的，则需要一个称为汇编程序的翻译程序将其翻译成目标程序后，才能在机器上执行。如果源程序是用某种高级语言编写的，则需要相应的解释程序或编译程序对其进行翻译。

解释程序也称为解释器，它或者直接解释执行源程序，或者将源程序翻译成某种中间表示形式后再执行。而编译程序（编译器）则是将源程序翻译成目标语言程序，然后在计算机上运行。这两种语言处理程序的根本区别是：在编译方式下，机器上运行的是与源程序等价的目标程序，源程序和编译程序都不再参与目标程序的执行过程；而在解释方式下，解释程序和源程序（或它的某种等价表示）要参与到程序的运行过程中，运行程序的控制权在解释程序。解释器在翻译源程序时不生成独立的目标程序，而编译器则需要将源程序翻译成独立的目标程序。

### 3. 程序设计语言的定义

一般情况下，程序设计语言的定义都会涉及语法、语义、语用和语境四个方面。

- 语法是指由程序设计语言的基本符号组成程序中的各个语法成分（包括程序）的一组规则，其中由基本符号构成符号（单词）的书写规则称为词法规则，由符号（单词）构成语法成分的规则称为语法规则。程序语言的语法可以通过形式语言进行描述。
- 语义是程序语言中按语法规则构成的各个语法成分的含义，可分为静态语义和动态语义。静态语义指编译时可以确定语法成分的含义，而运行时刻才能确定的含义是动态语义。一个程序的执行效果说明了该程序的语义，它取决于构成程序的各个组成部分的语义。
- 语用表示了构成语言的各个记号和使用者的关系，涉及符号的来源、使用和影响。
- 语言的实现则有个语境问题。语境是指理解和实现程序设计语言的环境，这种环境包括编译环境和运行环境。

### 4. 程序语言的发展概述

程序语言有交流算法和计算机实现的双重目的。现在的程序语言种类繁多，它们在实际应用上各有不同的侧重。

若一种程序语言不依赖于机器硬件，则称为高级语言；若程序语言能够应用于范围广泛的问题求解过程中，则称之为通用的程序设计语言。

FORTRAN 是第一个被广泛用来进行科学计算的高级语言。一个 FORTRAN 程序由一个主程序或一个主程序加若干个子程序组成。主程序及每一个子程序都是独立的程序单位，称为一个程序模块。在 FORTRAN 中，子程序是实现模块化的有效途径。

ALGOL60 主导了 20 世纪 60 年代程序语言的发展。它有严格的文法规则，采用巴克斯范式 (BNF) 来描述语言的语法。ALGOL 是一个分程序结构的语言。一般来说，一个 ALGOL 程序本身就是一个分程序。每个分程序由 `begin` 和 `end` 括起来，以说明分程序的范围和它所管辖的名字的作用域。分程序的结构可以是嵌套的，也就是说，分程序内可以包含别的分程序。过程也可以称为一个分程序。同一个名字在不同的分程序中可以代表完全不同的实体。如果一个名字在若干层嵌套分程序中被多次声明，则程序中该名字的使用由离使用点最近的内层声明来决定，即“最近嵌套原则”。此外，ALGOL 还提供了数组的动态声明和过程的递归调用。由于一个分程序只有在执行时才需要数据空间，执行完成后就释放所占用的空间。因此，分程序结构的主要优点是可以非常有效地使用存储器。另外，由于分程序结构的嵌套性，用一个栈来组织和管理整个程序运行时的数据空间是非常方便的。

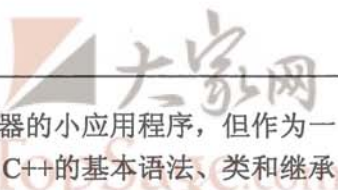
COBOL (COmmon Business Oriented Language) 是一种面向事务处理的高级语言。在企业管理中，数值计算并不复杂，但数据处理的信息量却很大。1959 年，由美国的一些计算机用户组织设计了专用于商务处理的计算机语言 COBOL，并于 1961 年由美国数据系统语言协会公布，经不断修改、丰富、完善和标准化，已发展了多种版本。COBOL 语言使用了三百多个英语保留字，采用了大量的普通英语词汇和句型。COBOL 语言的语法规则很严格，目前它主要应用于情报检索、商业数据处理等领域。

PASCAL 语言是一种结构化程序设计语言，由瑞士苏黎世联邦工业大学的沃斯 (N. Wirth) 教授研制，1971 年正式发表。PASCAL 是从 ALGOL60 衍生的，但功能更强且容易使用。PASCAL 语言在高校计算机软件教学中曾一直处于主导地位。后来的 PASCAL 语言中添加了并行控制结构，产生了并行 PASCAL。在 PASCAL 语言中分程序和过程这两个概念合二为一，统一为过程，而一个 PASCAL 程序本身可被看做是一个操作系统调用的过程。PASCAL 过程是可以嵌套和递归的。

C 语言是 20 世纪 70 年代发展起来的一种通用程序设计语言，它提供了一个丰富的运算符集合以及比较紧凑的语句格式。C 语言的主要特色是兼顾了高级语言和汇编语言的特点——简洁、丰富、可移植。C 与 UNIX 操作系统紧密相关，UNIX 操作系统本身及其上的许多软件都是用 C 编写的。C 提供了高效的执行语句，并且允许程序员直接去访问操作系统和底层硬件，这使得 C 在系统应用和实时处理中成为主要语言。

C++ 是在 C 语言的基础上于 20 世纪 80 年代发展起来的，与 C 兼容。在 C++ 中，最主要的是增加了类机制，使其成为一种面向对象的程序设计语言。





Java 产生于 20 世纪 90 年代，其目的是用于开发网络浏览器的小应用程序，但作为一种通用的程序设计语言，Java 也得到了广泛应用。Java 保留了 C++ 的基本语法、类和继承等概念，删掉了 C++ 中一些不好的特性。与 C++ 相比，Java 更简单，其语法和语义更合理。

### 5. 嵌入式程序设计语言

由于嵌入式系统自身的特点，因而对应用软件的开发和编程语言的选择提出了不同的要求。具体来说，需要考虑如下一些因素。

- 实时性：许多嵌入式系统要求具有实时处理的能力，这种实时性主要是靠软件层来体现的。软件对外部事件做出反应的时间必须要快，在某些情况下还要求是确定的、可重复实现的，不管系统当时的内部状态如何，都是可以预测的。
- 并发性：有些嵌入式系统要求支持多任务，能够处理并发事件。
- 有限的软硬件资源：在嵌入式系统当中，资源是很有限的，无论是处理器的运算速度、存储器的容量还是 I/O 设备的种类，都比不上通用的计算机，甚至连最基本的电力供应，在嵌入式系统中也是一项约束条件，因为许多系统都是采用电池供电。这就使得嵌入式软件在时间和空间上都受到了严格的限制。
- 涉及底层软件的开发：在无操作系统的情形下，嵌入式软件的开发是直接在硬件平台上进行的，需要直接对硬件进行控制；在有操作系统的情形下，需要先移植一个操作系统，并提供必要的 API 函数，然后在操作系统的基础上开发应用程序。但不管是哪一种情形，都需要对底层的软件和硬件进行操作，包括引导加载程序的编写、设备驱动程序的编写、对设备控制器的操作等。
- 需要交叉编译：嵌入式软件的开发环境与运行环境是不同的，需要交叉编译工具。

在这么多约束条件下，当我们在开发嵌入式软件时，就要非常讲究，编写出来的代码必须满足以下条件：

- 功能必须正确。
- 源代码要简洁、可读性好、可维护。
- 对于实时性要求比较高的代码，它的运行速度要足够快。
- 生成的目标代码要小，效率要高。

总之，在编写嵌入式软件时，要注意对执行时间、存储空间和开发/维护时间这三种资源的使用进行优化，也就是说，代码的执行速度要越快越好，系统占用的存储空间要越小越好，软件开发和维护的时间要越少越好。

与此相对应，在选择编程语言时，也必须满足以下一些条件：

- 简洁、实用。在语法上必须简洁、紧凑、表达能力强，在学习的时候比较容易入门，在使用的时候灵活、方便，功能强大。
- 编译出来的代码效率高、速度快。

- 具有较强的直接操作硬件的能力，包括与汇编语言的接口、对系统资源的直接访问、位操作、中断和异常事件处理等。
- 可移植性好，支持交叉编译。在许多硬件平台和操作系统上都有相应的编译器，编写出来的程序具有很好的可移植性，能在不同的硬件平台上运行。

在这些条件约束下，不是所有的编程语言都适合于嵌入式软件的开发。汇编语言与硬件的关系非常密切，效率最高，但是使用起来不太方便，程序开发和维护的效率比较低。而 C 语言是一种“高级语言中的低级语言”，它既具有高级语言的优点，又比较接近于硬件，凡是汇编语言能实现的功能，C 语言基本上都能实现，而且语言的效率也比较高。可以说，它满足以上的所有条件，所以 C 语言已成为嵌入式系统开发的最佳选择。

《嵌入式系统编程》(Embedded Systems Programming) 这份期刊在 1998 年刊登了一篇文章 Embedded Systems Programming: A 10-year retrospective，该文章做了一次调查，调查的题目是：在过去的一年中，你在嵌入式系统开发中曾经使用过哪一些编程语言。最后的统计结果见表 4-1。由此可见，选择 C 语言和汇编语言的人占绝大多数，远远超过了其他的编程语言。当然，该调查是在 1998 年之前进行的。近年来，在硬件上，嵌入式设备获得了迅猛的发展，无论是处理器的运算速度、存储器的容量还是 I/O 设备的种类都有很大的提升；在软件上，各种嵌入式软件开发工具不断涌现，各种高级语言的编译器也不断地改良。因此，越来越多的编程语言被应用到嵌入式系统的开发当中。

表 4-1 编程语言使用统计

嵌入式编程语言	使用人数
C 语言	81%
汇编语言	70%
C++ 语言	39%
Visual Basic	16%
Java 语言	7%

一个典型的嵌入式程序的生成过程如图 4-3 所示。

## 4.2.2 汇编语言

### 1. 基本原理

汇编语言是为特定计算机或计算机系统设计的面向机器的符号化程序设计语言。用汇编语言编写的程序称为汇编语言源程序。由于计算机不能直接识别和运行符号语言程序，所以需要专门的翻译程序——汇编程序进行翻译。用汇编语言编写程序时要遵循所用语言的规范和约定。

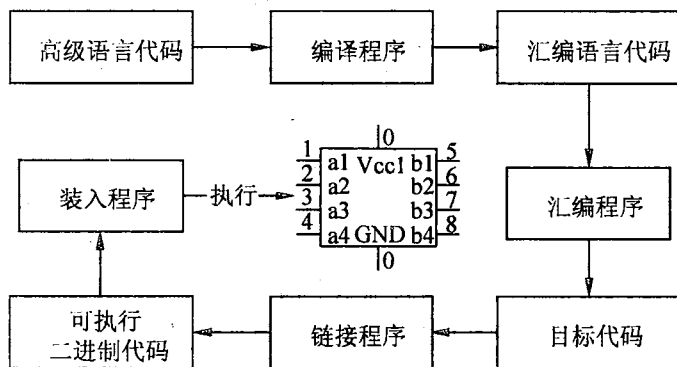


图 4-3 一个典型嵌入式程序的生成过程

汇编语言源程序由若干条语句组成。一般来说，在一个程序中可以有三类语句：指令语句、伪指令语句和宏指令语句。

#### (1) 指令语句

指令语句又称为机器指令语句，将其汇编后能产生相应的机器代码，这些代码能被 CPU 直接识别并执行相应的操作。基本的指令有 ADD、SUB 和 AND 等，书写指令语句时必须遵循指令的格式要求。

指令语句可分为传送指令、算术运算指令、逻辑运算指令、移位指令、转移指令和处理器控制指令等。

#### (2) 伪指令语句

伪指令语句指示汇编程序在对源程序进行汇编时完成某些工作。例如，给变量分配存储单元地址，给某个符号赋一个值等。伪指令语句与指令语句的区别是：伪指令语句经汇编后不产生机器代码，而指令语句经汇编后要产生相应的机器代码。另外，伪指令语句所指示的操作是在源程序被汇编时完成的，而指令语句的操作必须在程序运行时完成。

通常，汇编语言都应设立下列伪指令。

##### ① 常数定义伪指令

例如，在 ARM 汇编语言中定义常数的格式为：

```
x EQU 50
```

其中，EQU 是语句的记忆码，x 是用户定义的常数。这条语句的功能是定义标记符 x 的值为 50。

##### ② 存储定义伪指令

例如，ARM 汇编语言使用 DCB 来定义内存单元。

```
str DCB "this is a test"
```

这条语句分配了一片连续的字节存储单元并进行了初始化, `str` 表示被分配的存储区域的起始地址。

### ③ 汇编控制伪指令

用于控制汇编程序的执行流程。

例如, 在 ARM 汇编语言中, 常用的汇编控制伪指令包括下面几条。

**IF、ELSE、ENDIF:** 条件判断指令。

**WHILE、WEND:** 循环执行指令。

### ④ 开始伪指令

例如, 在 ARM 中可以使用 `ENTRY` 伪指令来指定汇编程序的入口点。

### ⑤ 结束伪指令

例如, 在 ARM 中 `END` 伪指令用于通知编译器已经到了源程序的结尾。

对于每一条汇编指令语句, 它由四个部分组成, 或者说, 被划分为四个区, 依次是标号区、操作码区、操作数区和注释区, 各个区域之间用确定的符号分隔开。标号区中的标号用于指示一条汇编指令语句, 它实际上代表该指令的内存单元地址; 操作码区是该语句的指令助记符, 它可以是机器指令助记符、伪指令码等; 操作数区指出本条汇编指令所操作的运算对象, 用寻址方式指定操作数的来源, 常用的是寄存器操作数和内存单元操作数。

## (3) 宏指令语句

在汇编语言中, 还允许用户将多次重复使用的程序段定义为宏。宏的定义必须按照相应的规定进行, 每个宏都有相应的宏名。在程序的任意位置, 若需要使用这段程序, 只要在相应的位置使用宏名, 即相当于使用了这段程序。因此, 宏指令语句就是宏的引用。

## 2. ARM 汇编语言

下面以 ARM 汇编语言为例, 对程序的基本格式及子程序间调用的格式进行一个简单介绍。

### (1) 汇编语言程序格式

ARM 汇编语言以段 (section) 为单位来组织源文件。段是相对独立的、具有特定名称的、不可分割的指令或数据序列。段又可以分为代码段和数据段, 代码段存放执行代码, 数据段存放代码运行时需要用到的数据。一个 ARM 源程序至少需要一个代码段, 大的程序可以包含多个代码段和数据段。

ARM 汇编语言源程序经过汇编处理后生成一个可执行的映像文件 (类似于 Windows 系统下的 EXE 文件)。该映像文件通常包括以下的三个部分:

- 一个或多个代码段, 代码段通常是只读的。

- 零个或多个包含初始值的数据段，这些数据段通常是可读写的。
- 零个或多个不含初始值的数据段，这些数据段被初始化为 0，通常是可读写的。

连接器根据一定的规则将各个段安排到内存的不同位置，源程序中相邻的段在可执行映像文件中不一定是相邻的。

下面通过一个简单的例子，说明 ARM 汇编语言源程序的基本结构。如图 4-4 所示，在一个 ARM 源程序中，使用 AREA 伪指令定义了一个段。AREA 表示一个段的开始，后面是这个段的名称及相关属性。在本例中定义了一个只读的代码段，其名称为 EXAMPLE1。

```
AREA  EXAMPLE1, CODE, READONLY
ENTRY
start
MOV r0, #10
MOV r1, #3
ADD r0, r0, r1
END
```

图 4-4 ARM 源程序基本结构

ENTRY 伪指令标识了程序的入口地址，即执行的第一条指令。在一个 ARM 程序中可以有多个 ENTRY，但至少要有有一个。一般来说，在初始化代码及异常中断处理程序中都包含了 ENTRY。如果程序包含了 C 代码，则 C 语言库文件的初始化部分也包含了 ENTRY。

本程序的主体部分实现了一个简单的加法运算。

END 伪指令告诉编译器源文件的结束。每一个汇编模块必须包含一个 END 指令，表明本模块的结束。

## (2) 汇编语言子程序调用

在 ARM 汇编语言中，子程序调用是通过 BL 指令来完成的。BL 指令的语法格式为：

BL subname ;subname 是调用的子程序的名称

BL 指令完成两个操作：将子程序的返回地址保存到 LR 寄存器中，然后将程序计数器 PC 的值设置为目标子程序的第一条指令的地址。这样，当需要从子程序返回时，只要把 LR 寄存器的值送到 PC 寄存器即可。另外，在子程序调用时，通常使用寄存器 R0~R3 来传递参数和返回结果。

图 4-5 所示是一个子程序调用的例子。子程序 DOADD 完成加法运算，操作数放在 R0

和 R1 寄存器中，结果放在 R0 中。

```
AREA EXAMPLE2, CODE, READONLY
ENTYR
start MOV r0, #10      ; 设置输入参数 R0
      MOV r1, #3       ; 设置输入参数 R1
      BL DOADD         ; 调用子程序 DOADD
doadd  ADD r0, r0, r1   ; 子程序
      MOV pc, lr       ; 从子程序中返回
      END
```

图 4-5 ARM 子程序调用的例子

### 4.2.3 面向过程的语言

#### 1. 概述

面向过程是指以一个具体的流程为单位，考虑它的实现办法。面向过程的语言，也叫命令式语言或强制式语言，在这种语言中，通过一系列可执行的运算及运算的先后次序来描述计算的过程。

命令式语言以冯·诺依曼计算机体系结构为背景，机器语言和汇编语言是最早问世的命令式语言，FORTRAN、ALGOL、COBOL、PASCAL 和 C 等高级语言也都属于此类语言。在这些语言中，变量对应于存储单元，对变量的访问就是对相应存储单元的访问；各个语句在程序中的顺序以及转向语句等控制语句则明确规定了机器的执行步骤。这就是冯·诺依曼体系结构在程序设计中的反映。

过程式语言程序的本质是重复地、按步地计算低级（非抽象）值并将之赋给变量（对象），这就迫使程序员去关心比较低级的细节，而这不适合于设计复杂的算法。因此，过去几十年来，过程式语言一直朝着隐蔽低级属性、提高程序的层次性和抽象性的方向发展。但无论它如何发展，都是基于冯·诺依曼体系结构的。例如，几乎所有的命令式语言都要涉及全局变量、传地址参数（或变量参数、引用参数等），它们一方面可以提高程序的执行效率，另一方面也降低了程序的层次性与抽象性，使程序难以编写、阅读、分析和修改，也使程序的正确性证明难以进行。随着在非冯·诺依曼体系结构、基础理论、元器件、人工智能与软件工程等方面的不断发展，已有一些非过程式语言问世，函数式程序设计语言与逻辑型程序设计语言就是其中的代表。但几乎每一种非过程式语言都会或多或少地掺入一些具有过程式语言特征的成分。而且，由于非过程式语言在效率、应用等方面存在的诸多问题，使得过程式语言在相当长的时期内仍会占据重要地位。



过程式语言的基本成分包括数据、运算、控制和函数等。

## 2. 数据成分

程序语言的数据成分指的是一种程序语言的数据类型。数据对象总是对应着应用系统中某些有意义的东西，数据表示则指定了程序中值的组织形式。数据类型用于代表数据对象，还用于在基础机器中完成对值的布局，同时还可用于检查表达式中对运算的应用是否正确。

数据是程序操作的对象，具有名称、类型、存储类别、作用域和生存期等属性，使用时要为它分配内存空间。

- 数据名称：由用户通过标识符来命名。标识符是由字母、数字和下划线组成的标记。
- 类型：说明数据占用内存的大小和存放形式。
- 存储类别：说明数据在内存中的位置和生存期。
- 作用域：说明可以使用数据的代码范围。
- 生存期：说明数据占用内存的时间范围。

从不同的角度可以对数据进行不同的划分。

### (1) 常量和变量

按照程序运行时数据的值能否改变，将数据分为常量和变量。程序中的数据对象可以具有左值和（或）右值。左值指存储单元（或地址、容器），右值指值（或内容）。变量具有左值和右值，在程序运行过程中其右值可以改变。常量只有右值，在程序运行过程中其右值不能改变。

### (2) 全局变量和局部变量

按数据的作用域范围，可以分为全局变量和局部变量。系统为全局变量分配的存储空间在程序运行过程中一般是不会改变的，而为局部变量分配的存储空间是动态改变的。

### (3) 数据类型

按照数据的组织形式的不同可将数据分为基本类型、用户定义类型、构造类型及其他类型。以 C 语言为例，它的数据类型如下所述。

- 基本类型：整型（int）、字符型（char）和实型（float、double）。
- 特殊类型：空类型（void）。
- 用户定义类型：枚举类型（enum）。
- 构造类型：数组、结构和联合。
- 指针类型。

## 3. 运算成分

程序语言的运算成分指明允许使用的运算符号及运算规则。大多数高级程序语言的基

本运算可以分为算术运算、关系运算和逻辑运算，有些语言还提供位运算。运算符的使用与数据类型密切相关。为了确保运算结果的唯一性，运算符要规定优先级和结合性，必要时还要使用圆括号。

#### 4. 控制成分

控制成分指明语言允许表达的控制结构，程序员使用控制成分来构造程序中的控制逻辑。理论上已经证明，可计算问题的程序都可以用顺序、选择和重复这三种控制结构来描述。

##### (1) 顺序结构

顺序结构用来表示一个计算操作序列。计算过程从所描述的第一个操作开始，按顺序依次执行后续的操作，直到序列的最后一个操作，如图 4-6 (a) 所示。在顺序结构内也可以包含其他控制结构。

##### (2) 选择结构

选择结构提供了在两种或多种分支中选择其中一个的逻辑。基本的选择结构是通过指定一个条件  $P$ ，然后根据条件的成立与否来决定控制流的走向，从两个分支中选择一个去执行，如图 4-6 (b) 所示。程序语言中通常还提供简化的选择结构，即只有一个分支选择，如图 4-6 (c) 所示。

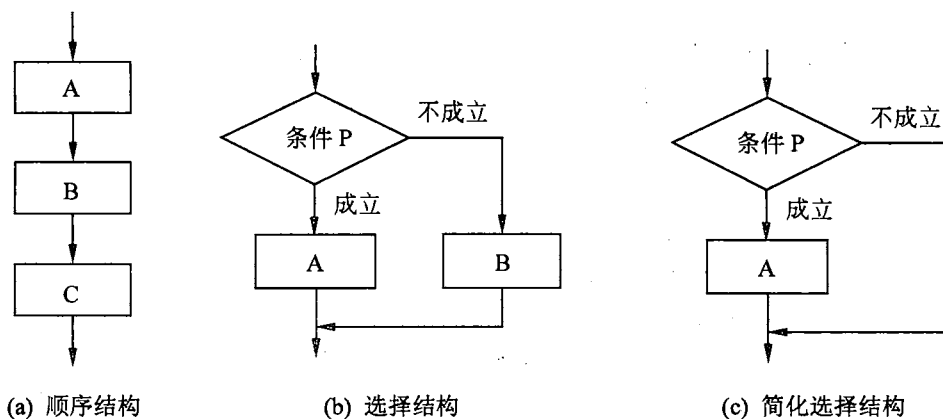


图 4-6 顺序结构与选择结构示意图

##### (3) 循环结构

循环结构描述了重复计算的过程，它通常由三部分组成：初始化、需要重复计算的部分和重复的条件。其中初始化部分有时在控制的逻辑结构中并无显式表示。重复结构主要有两种形式，**while** 型重复结构和 **do-while** 型重复结构。**while** 型重复结构的逻辑含义是先判断条件  $P$ ，若成立，则执行需要重复的程序块  $A$ ，然后再去判断重复条件；若不成立，



控制流就退出重复结构,如图 4-7 (a) 所示。**do-while** 型结构的逻辑含义是先执行需要重复执行的程序块 A,然后再判断条件 P,若成立则继续执行程序块 A;若不成立控制流就退出重复结构,如图 4-7 (b) 所示。

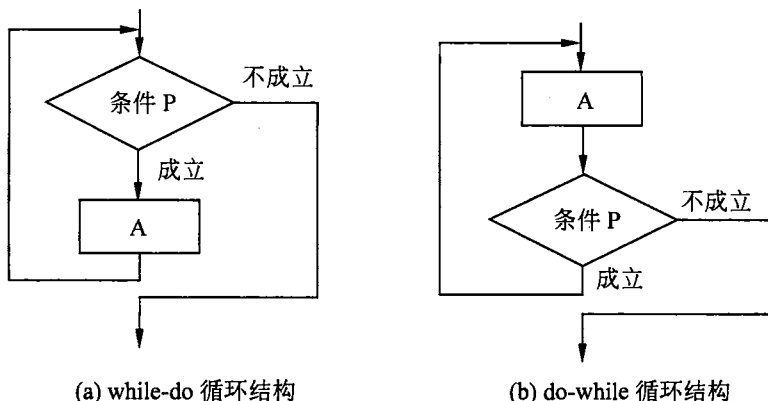


图 4-7 循环结构示意图

#### (4) C 语言提供的控制语句

##### ① 复合语句

复合语句用于描述顺序控制结构。复合语句是一系列用“{”和“}”括起来的声明和语句,其主要作用是将多条语句组合成一个可执行单元。语法上能出现语句的地方都可以使用复合语句。复合语句是一个整体,要么全部执行,要么一条语句也不执行。

##### ② if 语句和 switch 语句

if 语句实现的是双分支选择结构,其一般形式为:

if (表达式) 语句 1; else 语句 2;

其中,语句 1 和语句 2 可以是任何合法的 C 语句。

switch 语句描述了多分支的选择结构,其一般形式如图 4-8 所示。

在执行 switch 语句时,首先计算表达式的值,然后用所得的值与列举的常量表达式的值依次比较,若没有任何一个常量表达式能够匹配,则执行 default 的“语句  $n+1$ ”,然后结束 switch 语句。若此值与第  $i$  ( $i=1, 2, \dots, n$ ) 个常量表达式的值相同,则执行“语句  $i$ ”。然后,如果该语句的后面有 break 语句,则退出 switch 结构,否则就继续往下去执行语句  $i+1$ 、语句  $i+2$  等。

表达式可以是任何类型的,常用的是字符型或整数型表达式。多个常量表达式可以共用一个语句组,语句组可以包含任何可执行语句,且无须用大括号括起来。

```
switch (表达式)
{
    case 常量表达式 1: 语句 1;
    case 常量表达式 2: 语句 2;
    ...
    case 常量表达式 n: 语句 n;
    default: 语句 n+1;
}
```

图 4-8 switch 语句基本结构

### ③ 循环语句

C 语言提供了三种形式的循环语句，用于实现重复型的控制结构。

**while** 语句描述了先判断条件再执行循环体的控制结构，它的一般形式为：

**while** (条件表达式) 循环体语句；

其中，循环体语句是内嵌的语句，当循环体语句多于一条时，应使用一对大括号“{”和“}”把它们括起来。执行 **while** 语句时，先计算条件表达式的值，当其值为非 0 时，就执行循环体语句，然后重新计算条件表达式的值；否则就结束 **while** 语句的执行过程。

**do-while** 语句实现了先执行循环体再判断条件的控制结构，其一般格式为：

**do** 循环体语句 **while** (条件表达式)；

执行 **do-while** 语句时，先执行内嵌的循环体语句，然后再计算条件表达式的值，若其值为非 0，则再一次执行循环体语句，并计算条件表达式的值。直到条件表达式的值为 0 时，才结束 **do-while** 语句的执行过程。

**for** 语句的基本格式是：

**for** (表达式 1；表达式 2；表达式 3) 循环体语句；

**for** 语句的使用非常灵活，其内部的三个表达式都可以省略，但用于分隔三个表达式的分号“；”不能遗漏。在 **for** 语句的语义中，表达式 1 只计算一次，其作用是进行一些变量的初始化；表达式 2 的值是判断循环结束条件，如果省略，则认为该表达式的值为非 0，这意味着 **for** 语句将无法终止；表达式 3 一般用于修改循环变量的值。

## 5. 函数

### (1) 函数的使用

C 程序由一个或多个函数组成，每个函数都有一个名字，其中有且仅有一个名字为 **main** 的函数，作为程序运行时的起点。函数是程序模块的主要成分，它是一段具有独立功能的

程序。函数的使用涉及三个概念：函数定义、函数声明和函数调用。

① 函数定义

函数的定义描述了函数做什么和怎么做。它包括两部分：函数首部和函数体。函数首部说明了函数返回值的数据类型、函数的名字和函数运行时所需的参数及类型；函数所实现的功能在函数体中进行描述。

② 函数声明

函数应该先声明后引用。如果程序中对一个函数的调用位于该函数的定义之前，则应该在调用前对被调用的函数进行声明。函数声明定义了函数原型，其目的在于告诉编译器传递给函数的参数个数、类型以及函数返回值的类型，参数表中仅需要依次列出函数定义中的参数类型。函数原型可以使编译器检查源程序中对函数的调用是否正确。

③ 函数调用

当需要在一个函数（称为主调函数）中使用另一个函数（称为被调函数）实现的功能时，便以函数名字进行调用，称为函数调用。在使用一个函数时，只要知道如何调用就可以了，不需要关心被调用函数的内部实现。因此，主调函数需要知道被调函数的名字、返回值和需要向被调函数传递的参数（个数、类型和顺序）等信息。

(2) 函数调用的过程

当一个函数调用发生时，它的内部机理是什么，执行了哪些步骤？图 4-9 所示是一个程序在运行时，它的内存分布状况。

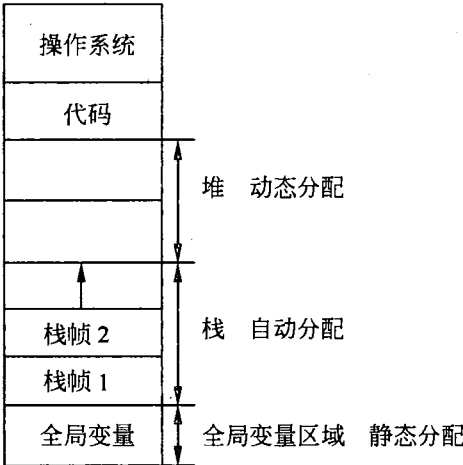


图 4-9 程序运行时的内存分布

当一个程序开始运行时，它的代码被装入到内存，保存在代码区，包括主函数和其他

函数的代码。另外，还有三块内存区域用来存放数据。第一块是全局变量区域，存放了程序当中的所有全局变量。由于全局变量的个数和大小是已知的，所以这一块区域所占用的内存大小在开始时即可确定下来，它们被称为是静态分配。位于此区域内的变量，它们在程序的整个运行过程当中，都一直存在，只有当整个程序运行结束了，这一块内存区域才会被释放。第二块区域是栈（stack）区域，它包含了所有的栈帧。所谓的栈帧（stack frame），就是在调用一个函数时，系统会自动地分配一块内存区域给这个函数，用来保存它的运行上下文、形参和局部变量等信息，这样的一块内存区域，就叫做一个栈帧。栈帧是在函数调用时分配的，当函数调用结束之后，相应的栈帧就会被释放。所以，对于一个函数的局部变量来说，只有当函数调用发生时，系统才会给这个函数的形参和局部变量分配存储空间；当函数调用结束后，这些局部变量就被释放掉了。另外，由于栈区域是由系统自动来分配的，用户并不需要去关心，所示也称为是自动分配。第三块区域是堆（heap）区域，它主要是用做动态分配的内存。

图 4-10 所示是函数调用的一个例子，它考察的是最简单的一种情形，即在整个程序当中，只有一个 main 函数。在这种情况下，当程序开始运行时，它就会被装入到内存。它的代码存放在内存的代码区域。由于在这个程序中定义了一个全局变量 z，所以就在内存的全局变量区域分配了一个存储单元给它，并且把它初始化为 0。接下来，系统就要调用主函数 main 去运行了，当这个函数调用发生时，系统就会在栈中给它分配一块内存空间，即一个栈帧，用来存放主函数当中所定义的局部变量，即 x 和 y。随后，程序计数器 PC 就跳转到主函数的第一条语句，开始执行。当 main 函数执行完后，首先要把它所占用的栈帧释放掉。对于任何一次函数调用来说，在函数调用结束后，都要把相应的栈帧释放掉，所以 x 和 y 这两个局部变量所占用的存储空间就被释放掉了，不能再访问了。接下来，由于 main 函数是一个特殊的函数，当它执行完之后，整个程序也就结束了。

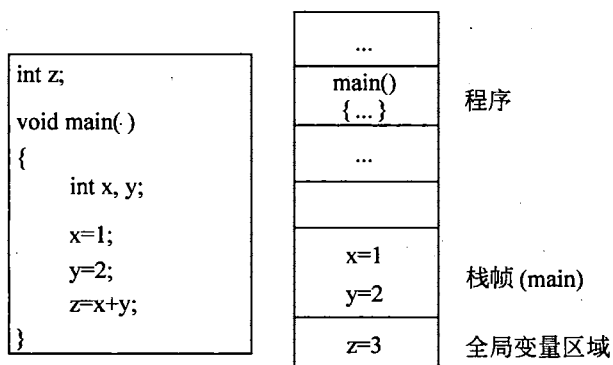


图 4-10 函数调用的例子



当一次函数调用发生时，它的执行过程可以归纳为以下 5 个步骤：

- ① 在内存的栈空间当中为其分配一个栈帧，用来存放该函数的形参变量和局部变量。
- ② 把实参变量的值复制到相应的形参变量中。
- ③ 控制流转移到该函数的起始位置。
- ④ 该函数开始执行。
- ⑤ 当这个函数执行完以后，控制流和返回值返回到函数调用点。

图 4-11 所示是变量的存储与作用域的一个例子。

```
/* 全局变量，固定地址，其他源文件可见 */
int global_static;

/* 静态全局变量，固定地址，但只在本文件中可见 */
static int file_static;

/* 函数参数：位于栈帧当中，动态创建，动态释放 */
int foo (int auto_param)
{
    /*静态局部变量，固定地址，只在本函数中可见 */
    static int func_static;

    /* 普通局部变量，位于栈帧当中，只在本函数可见 */
    int auto_i, auto_a[10];

    /* 动态申请的内存空间，位于堆当中 */
    double *auto_d = malloc (sizeof (double) *5);

    return auto_i;
}
```

图 4-11 变量的存储与作用域的例子

#### 4.2.4 面向对象的语言

##### 1. 面向对象的基本概念

Peter Coad 和 Edward Yourdon 提出用下面的等式来识别面向对象方法：

面向对象 = 对象 (object)  
+ 分类 (classification)  
+ 继承 (inheritance)  
+ 通过消息的通信 (communication with messages)

可以说,采用这四个概念开发的软件系统是面向对象的。

### (1) 对象

在面向对象的系统中,对象是基本的运行时的实体,它既包括数据(属性),也包括作用于数据的操作(行为),所以一个对象把属性和行为封装为一个整体。封装是一种信息隐蔽技术,它的目的是使对象的使用者和生成者分离,使对象的定义和实现分开。从程序设计者来看,对象是一个程序模块;从用户来看,对象为他们提供了所希望的行为。在对象内的操作通常叫做方法(method)。一个对象通常可由对象名、属性和操作三部分组成。

在现实世界中,每个实体都是对象,如学生、汽车、电视机、空调等都是现实世界中的对象。每个对象都有它的属性和操作,如电视机有颜色、音量、亮度、灰度、频道等属性,可以有切换频道、增大/降低音量等操作。电视机的属性值表示了电视机所处的状态,而这些属性只能通过其提供的操作来改变。电视机的各个组成部分,如显像管、电路板、开关等都封装在电视机的机箱中,人们不知道也不关心电视机是如何实现这些操作的。

### (2) 消息

对象之间进行通信的一种构造叫做消息。当一个消息发送给某个对象时,包含要求接收对象去执行某些活动的信息,接收到信息的对象经过解释,然后予以响应。这种通信机制叫做消息传递。发送消息的对象不需要知道接收消息的对象如何对请求予以响应。

### (3) 类

一个类定义了一组大体上相似的对象,一个类所包含的方法和数据描述了一组对象的共同行为和属性。把一组对象的共同特征加以抽象并存储在一个类中的能力,是面向对象技术最重要的一点。而是否建立了一个丰富的类库,是衡量一个面向对象程序设计语言成熟与否的重要标志。

类是在对象之上的抽象,对象是类的具体化,是类的实例(instance)。在分析和设计时,我们通常把注意力集中在类上,而不是具体的对象。我们也不必对每个对象逐个定义,只需对类做出定义,而对类的属性进行不同的赋值即可得到该类的对象实例。

有些类之间存在一般和特殊关系,即一些类是某个类的特殊情况,某个类是一些类的一般情况,这是一种“is-a”关系,即特殊类是一种一般类。例如,“汽车”类、“轮船”类、“飞机”类都是一种“交通工具”类。特殊类是一般类的子类,一般类是特殊类的父类。同样,“汽车”类还可以有更特殊的类,如“轿车”类、“货车”类等,在这种关系下形成一种层次的关联。

通常把一个类和这个类的所有对象称为“类及对象”或对象类。

### (4) 继承

继承是父类和子类之间共享数据和方法的机制。这是类之间的一种关系,在定义和实现一个类的时候,可以在一个已经存在的类的基础上来进行,把这个已经存在的类所定义

的内容作为自己的内容，并加入若干新的内容。图 4-12 所示表示了父类 A 和它的子类 B 之间的继承关系。

一个父类可以有多个子类，这些子类都是父类的特例，父类描述了这些子类的公共属性和操作。一个子类已继承它的父类（或祖先类）中的属性和操作——这些属性和操作在子类中不必再次定义，且子类中还可以定义自己的属性和操作。

图 4-12 中的 B 只从一个父类 A 得到继承，这叫“单重继承”。如果一个子类有两个或多个父类。则称为“多重继承”。

### (5) 多态

在收到消息时，对象要予以响应。不同的对象收到同一消息可以产生完全不同的结果，这一现象叫做多态 (polymorphism)。在使用多态的时候，用户可以发送一个通用的消息，而实现的细节则由接收对象自行决定，这样，同一消息就可以调用不同的方法。

多态的实现受到继承的支持，利用类的继承的层次关系，把具有通用功能的消息存放在高层次，而不同的实现这一功能的行为放在较低层次。在这些层次上生成的对象能够给通用消息以不同的响应。

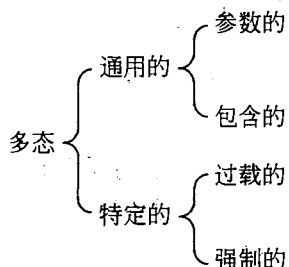


图 4-13 多态的形式

多态有几种不同的形式，Cardelli 和 Wegner 把它分为四类，如图 4-13 所示。其中，参数多态和包含多态称为通用的多态，过载多态和强制多态称为特定的多态。

参数多态是应用比较广泛的多态，被称为最纯的多态；包含多态在许多语言中都存在，最常见的例子就是子类型化，即一个类型是另一个类型的子类型；过载 (overloading) 多态是同一个变量被用来表示不同的功能，然后通过上下文来决定一个名所代表的是其中的哪个功能。

### (6) 动态绑定 (dynamic binding)

绑定是一个把过程调用和响应调用所需要执行的代码加以结合的过程。在一般的程序设计语言中，绑定是在编译时进行的，叫做静态绑定。动态绑定则是在运行时进行的，因此，一个给定的过程调用和代码的结合是到调用发生时才进行的。

动态绑定是同类的继承以及多态相联系的。在继承关系中，子类是父类的一个特例，所以父类对象可以出现的地方，子类对象也可以出现。因此在运行过程中，当一个对象发送消息请求服务时，要根据接收对象的具体情况将请求的操作与实现的方法进行连接，即

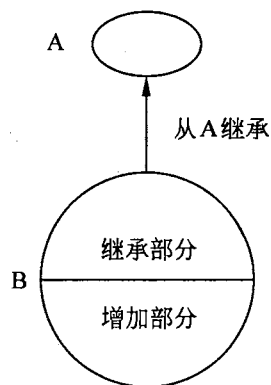


图 4-12 类的继承关系

动态绑定。

## 2. 面向对象的程序设计语言

面向对象程序设计 (Object-Oriented Programming, OOP) 的实质是选用一种面向对象程序设计语言 (Object-Oriented Programming Language, OOPL), 采用对象、类及其相关概念所进行的程序设计。

在程序设计语言中引入面向对象的概念实际上不是最近几年才有的, 其根源可以追溯到 20 世纪 60 年代后期, 那时 Kristen Nygaard 和 Ole-Johan Dahl 在挪威计算中心开发出了—种叫做 Simula 67 的语言。这种语言在用于模拟的语言 Simula 的基础上, 第一次引入了类、协同程序 (coroutines) 和子类的概念, 非常类似于今天的面向对象程序设计语言。

20 世纪 70 年代中期, 在 Xerox 公司的 Palo Alto 研究中心 (Xerox PARC) 中产生了第一个完整、健全的面向对象程序设计语言 Smalltalk, 以后又进行了改进和扩充, 形成了 Smalltalk-80。这种语言中的每一个元素都分别是一个对象, 从语言本身的实现、程序设计环境, 到所支持的程序设计风格, 都是面向对象的。Simula 67 和 Smalltalk 的开发, 是目前面向对象程序设计和面向对象程序设计语言研究中大多数工作的起点。

到目前为止, 已经有大约一百多种面向对象的和基于对象的程序设计语言。其中, 比较流行或独特的 OOPLs 包括 Smalltalk、Eiffel、C++ 和 Java。

### (1) Smalltalk

Smalltalk 语言起源于 20 世纪 60 年代末期, 主要设计者 Alan Kay 还在犹他州大学念研究生时, 就认识到可以用 Simula 语言中的概念来充实他在图形方面的研究工作。当他到 Xerox PARC 工作后, 将这些早期思想发展为基于图形的交互式个人工作站概念, 并萌发了设计—种新语言来支持这些概念的想法。

Smalltalk 在 Xerox PARC 历经了多次重大修改, 最终形成了 Smalltalk-80。事实上, Smalltalk 并不是一种单纯的程序设计语言, 而是反映面向对象程序设计思想的程序设计环境。这个系统在系统本身的设计中强调了对象概念的归—性, 引入了类、方法、实例等概念和术语, 应用了单重继承和动态绑定, 成为 OOPL 发展过程中的—个引人注目的里程碑。

在 Smalltalk-80 中, 除了对象之外没有其他形式的数据, 对—个对象的唯一操作就是向它发消息。在这种语言中, 连类也被看成是对象——类是元类的实例, 因此, 定义—个类就是向元类发—条消息, 请求元类执行它的 new 操作来生成—个实例, 也就是生成这个类, 而消息中的参数, 就是关于这个类的说明。

### (2) Eiffel

Eiffel 语言是继 Smalltalk-80 之后的另—个“纯”OOPL。这种语言是由 OOP 领域中著名的专家 Bertrand Meyer 等人于 20 世纪 80 年代后期在 ISE 公司 (Interactive Software Engineering Inc.) 开发的, 它的主要特点是全面的静态类型化、有大量的开发工具、支持



多继承。在众多的 OOPLs 中, Eiffel 是较早对多重继承提供支持的, 它的一些实现策略, 如同名冲突处理、异常处理等, 已经对后来的 OOPLs 的设计和实现产生了影响。

### (3) C++

C++语言是一种面向对象的强类型化语言, 由 AT&T 的 Bell 实验室于 1980 年推出, 目前已被国外许多主要的计算机和软件生产企业选为替代 C 语言或与 C 语言并存的基本开发工具。

C++语言是 C 语言的一个向上兼容的扩充, 而不是一种新语言。C++是一种支持多范型的程序设计语言, 它既支持面向对象的程序设计, 也支持面向过程的程序设计。它融合了 Simula 67 的几种面向对象机制——类、继承、虚拟函数等, 借鉴了 ALGOL 68 中的操作符过载、变量声明位置不受限制等特性, 形成了一种“比 Smalltalk 更接近于机器、比 C 语言更接近于问题”的 OOPL。

C++支持基本的面向对象概念: 对象、类、方法、消息、子类和继承; 它同时支持静态类型和动态类型; C++完全支持多继承, 并通过使用 try/throw/catch 模式提供了一个完整的异常处理机制。

C++不提供自动的无用存储单元收集, 这必须通过程序员来实现, 或者通过编程环境提供合适的代码库来予以支持。

C++语言强大的类、继承等功能更便于实现复杂的程序功能, 目前在嵌入式系统设计中得到了广泛的应用, 如 GNU C++。它的核心语言特性同 C 完全一样, 但是 C++提供了更好的数据抽象和面向对象形式的编程功能。这些新的特性对软件开发人员非常有帮助, 但是部分特性会降低可执行程序的性能, 所以 C++在一些较大的开发团队中用的最为普遍, 在那里 C++对程序员的帮助要比程序效率的损失更为重要。但与 C 相与, C++的目标代码往往比较庞大和复杂, 在嵌入式系统应用中应充分考虑这一因素。

C++语言为了支持复杂的语法, 在代码生成效率方面难免有所下降。为此, 1995 年初在日本成立的 Embedded C++技术委员会经过几年的研究, 针对嵌入式应用制定了减小代码容量的 EC++标准。EC++保留了 C++的主要优点, 提供对 C++的向上兼容性, 并满足嵌入式系统设计的一些特殊要求。在嵌入式高级语言编译器方面处于领先地位的 Tasking 公司, 是 EC++技术委员会成员之一, 也是最先推出 EC++产品的公司。

C/C++/EC++被引入嵌入式系统, 使得嵌入式开发同个人计算机、小型机等之间在开发上的差别正在逐渐消除, 软件工程中的很多经验、方法乃至库函数可以移植到嵌入式系统中。在嵌入式开发中采用高级语言, 还使得硬件开发和软件开发可以分工——从事嵌入式软件开发不再必须精通系统硬件和相应的汇编语言指令集。

### (4) Java

Java 语言起源于 Oak 语言, Oak 语言被设计成能运行在设备 (如微波炉、摄像机等)

的嵌入芯片上。

Java 运行时被编译成伪代码 (P 码或者字节代码), 所以需要一个虚拟机来对其进行解释。Java 的虚拟机在几乎每一种平台上都可以运行。这实质上使得开发是与机器独立无关的, 并且提供了通用的可移植性。

Java 把类的概念和接口 (interface) 的概念区分开来, 并试图通过只允许接口的多继承来克服多继承的危险。

Java 的异常处理机制与 C++ 的 try/throw/catch 相类似, 但更加严密。在 Java 中, 通过声明轻型线程来处理并行性, 这些线程通过副作用和同步协议进行通信。

Java Beans 是组件, 即类和其所需资源的集合, 它们主要被设计用来提供定制的 GUI 小配件。

Java 有自己的对象请求代理技术 RMI (远程方法调用), 这使得应用能够跨越网络来调用在其他 Java 应用程序内的方法。从第 2 版起, Java 也包含了一个兼容 CORBA 的、语言独立的对象请求代理, Java 还包含了用于管理与关系数据库进行交互的类 JDBC 和用于基本图形的类。

随着不断增长的市场需求, 很多嵌入式设备必须适应网上交流的需要。为了迎合此要求, 考虑到开发 Internet 应用程序的便利, 众多开发者都发现使用 Java 语言或 C# 是有意义的。另外, 随着内存及 32 位处理器价格的下降, 最初在嵌入系统使用 Java 或 C# 太昂贵的问题不再有了, 使用的成本开始减少。于是, Java 或 C# 语言在嵌入式领域迎来了新的机遇。

## 4.2.5 汇编、编译与解释程序的基本原理

### 1. 汇编程序基本原理

汇编程序是最早也是最成熟的一种系统软件, 它的功能是将汇编语言源程序翻译成机器语言程序。此外, 它还具有一些其他功能, 例如, 根据用户指定自动分配存储区域, 包括代码区、数据区、暂存区等; 自动地把各种进位制数转换成二进制数、把字符转换成 ASCII 码、计算表达式的值等; 自动对源程序进行检查, 给出错误信息 (如非法格式、未定义的助记符、标号, 漏掉操作数等)。

汇编程序可以用汇编语言来编写, 也可以用各种高级语言编写。汇编程序的类型很多, 但主要功能都一致, 只是附加功能各有不同。

汇编程序的类型主要有以下几种。

- 交叉汇编程序: 运行汇编程序的计算机与汇编后目标程序所运行的机器是不同的。例如, 汇编程序本身在 PC 机上运行, 而汇编成的目标程序是在 MSC-51 系列单片机上执行的。
- 驻留汇编程序: 运行这种汇编程序的计算机就是执行目标程序的计算机。

- 宏汇编程序：它允许把一组指令序列定义为一条宏指令，有宏汇编功能。

由于汇编指令中形成操作数地址的部分可能出现后面才会定义的符号，所以汇编程序一般至少需要两次扫描源程序才能完成翻译过程。

第一次扫描的主要工作是定义符号的值并创建一个符号表 ST。ST 记录了汇编时所遇到的符号的值。另外，有一个固定的机器指令表 MOT1，其中记录了每条机器指令的记忆码和指令的长度。在汇编程序翻译源程序的过程中，为了计算各汇编语句中标号的地址，需要设立一个位置计数器或单元地址计数器 LC (Location Counter)，其初值一般为 0。在扫描源程序时，每处理完一条机器指令或一条与存储分配有关的伪指令（如定义常数语句，定义储存语句），LC 的值就增加相应的长度。这样，在汇编过程中，LC 的内容就是下一条被汇编的指令的偏移地址。若正在被汇编的语句有标号，则该标号的值就取 LC 的当前值。LC 的基本原理如图 4-14 所示。

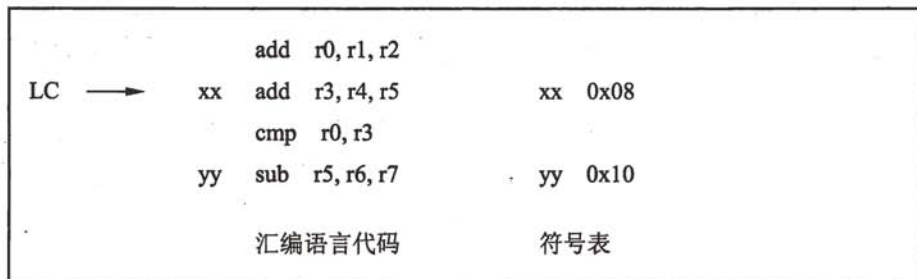


图 4-14 LC 的示意图

此外，在第一次扫描中，还需要对与定义符号值有关的伪指令进行处理。为了叙述方便，不妨设立伪指令表 POT1。POT1 表的每一个元素只有两个域：伪指令助记符和相应的子程序入口。下面的步骤描述了汇编程序第一次扫描源程序的过程。

(1) 单元计数器 LC 置初值 0。

(2) 打开源程序文件。

(3) 从源程序中读入第一条语句。

(4) while (若当前语句不是 END 语句)

{

if (当前语句有标号) 则将标号和单元计数器 LC 的当前值填入符号表 ST;

if (当前语句是可执行的汇编指令语句) 则查找 MOT1 表，获得当前指令的长度 K，并令  $LC = LC + K$ ;

if (当前指令是伪指令) 则查找 POT1 表并调用相应的子程序;

if (当前指令的操作码是非法记忆码) 则调用出错处理子程序;

从源程序中读入下一条语句。

}

#### (5) 关闭源程序文件。

第二次扫描的任务是产生目标程序。除了使用前一次扫描所生成的符号表 ST 外, 还要使用机器指令表 MOT2。MOT2 表中的元素有机器指令助记符、机器指令的二进制操作码 (binary-code) 以及格式 (type) 和长度 (length)。此外, 还要设立一个伪指令表 POT2, 供第二次扫描时使用。POT2 的每一个元素仍有两个域: 伪指令记忆码和相应的子程序入口。与第一次扫描不同的是: 在第二次扫描中, 对伪指令有着完全不同的处理。

在第二次扫描中, 可执行汇编语句应被翻译成对应的二进制代码机器指令。这一过程涉及两个方面的工作: 一是把机器指令助记符转换成二进制机器指令操作码, 这可通过查找 MOT2 表来实现; 二是求出操作数区各操作数的值 (用二进制表示)。在此基础上, 就可以装配出用二进制代码表示的机器指令。

汇编程序最简单的形式是假定汇编语言程序的地址已被编程者指明, 在这种程序中的地址称作绝对地址。例如, 在 ARM 中, 采用 ORG 语句说明程序的起始地址。

但在许多情况下, 程序可能由很多个文件生成, 如果采用绝对地址方式, 编程者必须在汇编前为每一个模块确定其在内存中的长度以及它们连接到程序的次序, 这种工作方式对于大程序或采用预汇编的库例程的情况是无法想象的。因此大多数汇编程序允许使用相对地址方式, 在文件开始处指明以后将被计算的汇编语言模块始址, 模块中的地址将从相对于该模块的开始处被计算。连接程序操作由汇编程序产生的目标文件, 并负责将相对地址转化为绝对地址, 生成文件间必要连接, 将其装订在一起。

连接程序分两个阶段进行。首先, 决定每一目标文件开始的绝对地址。目标文件被载入的顺序由用户给定, 或者通过装入程序运行时指明的参数, 或者通过创建装入映像文件产生文件置入内存顺序。给出文件载入内存顺序和每一目标文件的长度后, 就很容易计算出每个文件的起始地址。第二阶段, 装入程序把所有目标文件符号表合并为单独的一个大表, 再编辑目标文件, 变相对地址为绝对地址。

在嵌入式系统中控制代码模块在何处载入内存很重要, 某些数据结构和指令 (例如中断管理指令) 必须被置入规定的存储单元中运行, 不同类型的存储器可能被置入不同的地址范围。例如, 如果在一些位置是 EPROM, 其他位置是 DRAM, 这时就必须确定写入位置是在 DRAM 单元中。

## 2. 编译程序基本原理

编译程序的功能是把某些高级语言编写的源程序翻译成与之等价的目标语言程序。在实现一个嵌入式系统时, 经常需要控制中断处理的指令顺序、内存中数据和指令的位置等, 所以理解一个高级语言如何被翻译为机器指令是很有用的, 能够帮助编程者更好地利用编

译程序的特性,生成高质量的目标代码。另外,许多嵌入式系统对性能的要求较高,所以通过理解代码的产生过程,将有助于编程语言的选择,即何时采用高级语言、何时采用汇编语言。

编译程序的工作过程一般可以分为六个阶段,如图 4-15 所示。每个阶段的操作在逻辑上是紧密相连的,将源程序从一种表示形式转换成另一种表示形式。当然,在实际的编译器中,可能会将其中的某些阶段结合在一起进行处理。

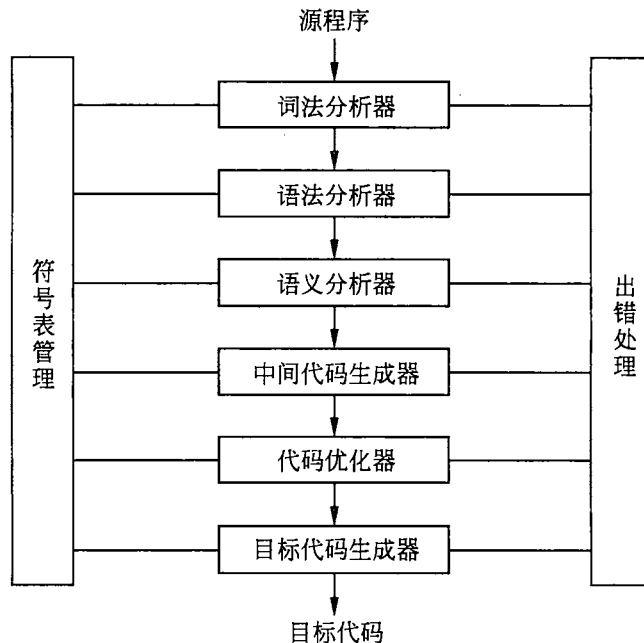


图 4-15 编译器的工作阶段示意图

下面简要介绍各个阶段实现的主要功能。

#### (1) 词法分析阶段

源程序可以被简单地看成是一个多行的字符串。词法分析阶段是编译过程的第一阶段,这个阶段的任务是对源程序从前到后(从左到右)逐个字符进行扫描,从中识别出一个个“单词”符号。“单词”符号是程序设计语言的基本语法单位,如关键字(或称保留字)、标识符、常数、运算符和分隔符(如标点符号、左右括号)等。词法分析程序输出的“单词”常采用二元组的方式,即单词类别和单词自身的值。

词法分析过程依据的是语言的词法规则,即描述“单词”结构的规则。例如,某 C 语言源程序中的一条声明语句和赋值语句:

```
float x, y;
```

```
x = y * 60
```

词法分析阶段将把构成这两条语句的字符串分割成如下的单词序列：

- ① 保留字 float ② 标识符 x ③ 逗号 , ④ 标识符 y  
⑤ 分号 ; ⑥ 标识符 x ⑦ 赋值号 = ⑧ 标识符 y  
⑨ 乘号 \* ⑩ 整常数 60

对于标识符  $x$  和  $y$ ，其单词类别都是  $id$ （标识符），字符串“ $x$ ”和“ $y$ ”都是单词的值，而对于单词 60，整常数是该单词的类别，60 是该单词的值。这里用  $id1$  和  $id2$  分别代表  $x$  和  $y$ ，强调标识符的内部标识由于组成该标识符的字符串不同而有所区别。经过词法分析后，声明语句 `float x, y` 被表示为 `float id1, id2`；赋值语句 `x = y * 60` 被表示为 `id1 = id2 * 60`。

## （2）语法分析阶段

语法分析的任务是在词法分析的基础上，根据语言的语法规则将单词符号序列分解成各类语法单位，如“表达式”、“语句”和“程序”等。语法规则就是各类语法单位的构成规则。通过语法分析，确定整个输入串是否构成一个语法上正确的程序。如果源程序中没有任何语法错误，语法分析后就能正确地构造出其语法树，否则就指出语法错误，并给出相应的诊断信息。例如，对于 `id1 = id2 * 60` 进行语法分析后形成的语法树如图 4-16 所示。

词法分析和语法分析本质上都是对源程序的结构进行分析。

## （3）语义分析阶段

语义分析阶段主要检查源程序是否存在语义错误，并收集类型信息供后面的代码生成阶段使用。只有语法和语义都正确的源程序才能翻译成正确的目标代码。

语义分析的一个主要工作是进行类型分析和检查。程序语言中的数据类型一般包含两个方面的内容。类型的载体及其上的运算。例如，整数取余只能对整型数据进行运算，若其运算对象中有浮点数，就认为是类型不匹配的错误。

在确认源程序的语法和语义正确之后，就可对其进行翻译并改变源程序的内部表示。对于声明语句，需要记录所遇到的符号的信息，所以应对符号表进行填查操作。在图 4-17 (a) 所示的符号表中，每一行存放一个符号的信息。第一行存放标识符  $x$  的信息，其类型为 `float`，为它分配的地址是 0；第二行存放  $y$  的信息，它的类型是 `float`，为它分配的地址是 4。因此，在这种语言中，为一个 `float` 型数据分配的存储空间是四个存储单元。对于可执行语句，则检查结构合理的表达式是否有意义。对 `id1 = id2 * 60` 进行语义分析后的语法树如图 4-17 (b) 所示，其中增加了一个语义处理节点 `inttoreal`，该运算用于将一个整型数

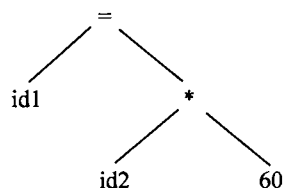


图 4-16 语法树示意图

转换为浮点数。

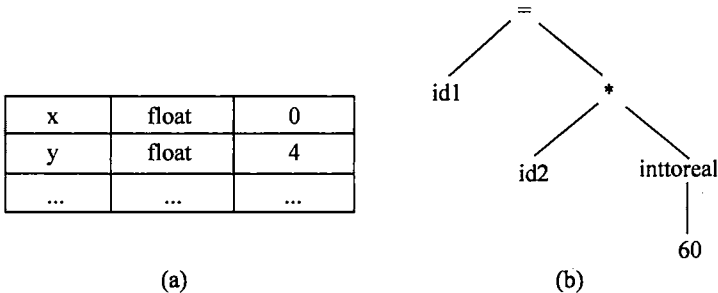


图 4-17 语义分析后的符号表和语法树示意图

(4) 中间代码生成阶段

中间代码生成阶段的工作是根据语义分析的输出生成中间代码。“中间代码”是一种简单且含义明确的记号系统，可以有若干种形式，它们的共同特征是与具体的机器无关。中间代码的设计原则主要有两点：一是容易生成；二是容易被翻译成目标代码。最常用的一种中间代码是与汇编语言的指令非常相似的三地址码，其实现方式常采用四元式。四元式的形式为：

(运算符，运算对象 1，运算对象 2，运算结果)。

例如，对于语句 `id1 = id2 * 60`，可生成以下四元式序列：

- ① (inttoreal, 60, -, t1)
- ② (\*, id2, t1, t2)
- ③ (=, t2, -, id1)

其中，t1 和 t2 是编译程序生成的临时变量，用于存放临时的运算结果。

语义分析和中间代码生成所依据的是语言的语义规则。

(5) 代码优化阶段

优化是一个编译器的重要组成部分，由于编译器将源程序翻译成中间代码的工作是机械的、按固定模式进行的，因此，生成的中间代码往往在时间上和空间上都有很大的浪费。当需要生成高效的目标代码时，就必须进行优化。优化过程可以在中间代码生成阶段进行，也可以在目标代码生成阶段进行。由于中间代码是不依赖于具体机器的，此时所做的优化一般建立在对程序的控制流和数据流分析的基础之上，与具体的机器无关。优化所依据的原则是程序的等价变换规则。

(6) 目标代码生成阶段

目标代码生成是编译器工作的最后一个阶段。这一阶段的任务是把中间代码变换成特定机器上的绝对指令码、可重定位的指令代码或汇编指令代码。这个阶段的工作与具体的

机器密切相关。

### (7) 符号表管理

符号表的作用是记录源程序中各个符号的必要信息,以辅助语义的正确性检查和代码生成。在编译过程中需要对符号表进行快速有效地查找、插入、修改和删除等操作。符号表的建立可以始于词法分析阶段,也可以放到语法分析和语义分析阶段,但符号表的使用有时会延续到目标代码的运行阶段。

### (8) 出错处理

用户编写的源程序不可避免地会有一些错误,这些错误大致可分为静态错误和动态错误。动态错误也称为动态语义错误,指程序中包含的逻辑错误。这种错误发生在程序运行时,例如,变量取0时作除数、数组下标越界访问等。静态错误是指编译阶段发现的程序错误,可分为语法错误和静态语义错误,如单词拼写错误、标点符号错误、表达式中缺少操作数、括号不匹配等有关语言结构上的错误称为语法错误,而语义分析时发现的运算符与运算对象类型的不匹配等错误属于静态语义错误。

在编译时发现程序中的错误后,编译程序应采用适当的策略跳过它们,使得分析过程能够继续下去,以便在一次编译过程中尽可能多地找出程序中的错误。

编译器的各个阶段逻辑上可以划分为前端和后端两部分。前端包括从词法分析到中间代码生成各阶段的工作,后端包括中间代码优化、目标代码的生成与优化等阶段。这样,以中间代码为分水岭,把编译器分成了与机器有关的部分和与机器无关的部分。如此一来,对于同一种程序语言的编译器,在开发出一个前端之后,就可以针对不同的机器开发相应的后端,前后端有机结合后就形成了该语言的一个编译器。当语言有改动时,只会涉及前端部分的维护。对于不同的程序语言,分别设计出相应的前端,然后将各个语言的前端与同一个后端相结合,就可得到各个语言在某种机器上的编译器。

## 3. 解释程序基本原理

解释程序是另一种语言处理程序,在词法、语法和语义分析方面与编译程序的工作原理基本相同,但在运行用户程序时,它直接执行源程序或源程序的内部形式。因此,解释程序不产生源程序的目标程序,这是它同编译程序的主要区别。图4-18所示为解释程序实现高级语言的三种方式。

源程序被直接解释执行的处理方式如图4-18中的标记A所示。这种解释程序对源程序逐个字符进行检查,然后执行程序语句规定的动作。例如,如果扫描到字符串序列:

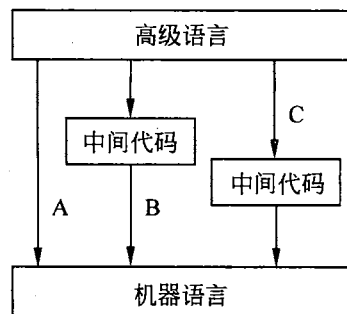


图4-18 解释程序类型示意图





## GOTO L

解释程序就开始搜索源程序中标号 L 后面紧跟冒号“:”的出现位置。这类解释程序通过反复扫描源程序来实现程序的执行,运行效率很低。

解释程序也可以先将源程序翻译成某种中间代码形式,然后对中间代码进行解释,实现用户程序的运行。这种翻译方式如图 4-18 中的标记 B 和 C 所示。通常,在中间代码和高级语言的语句间存在一一对应的关系。解释方式 B 和 C 的不同之处在于中间代码的级别,在方式 C 下,解释程序采用的中间代码更接近于机器语言。在这种实现方案中,高级语言和低级中间代码存在着 1:n 的对应关系。

解释程序通常可以分为两部分:第一部分是分析部分,包括通常的词法分析、语法分析和语义分析程序,经语义分析后把源程序翻译成中间代码,中间代码常采用逆波兰表示形式;第二部分是解释部分,用来对第一部分所生产的中间代码进行解释执行。

## 4.3 嵌入式软件开发环境

在早期的嵌入式开发中,大多数嵌入式软件的开发是直接建立在硬件平台的基础上进行的,采用处理器的汇编语言进行编程,直接对各种硬件设备进行控制和访问。用户除了要编写具体的应用程序外,还要编写各种监控程序和调试工具软件来构建相应的调试环境。尤其是对于多任务和实时性处理,必须编写出性能优化的系统软件,根据各个任务的重要性进行统筹兼顾和合理调度,以确保每个任务能及时执行,满足系统的实时性要求。随着嵌入式产品规模越来越大,功能越来越复杂,这种手工作坊式的开发方式越来越不能满足需要。

面对这些困难,人们提出以下要求。

- 嵌入式系统开发需要专门的开发工具和调试环境,支持多种软硬件平台,提供一种高级编程语言,如 C 或 C++;由开发工具提供针对多种处理器的编译系统,使开发代码易于移植扩充;调试环境提供的调试手段丰富,易于发现问题。
- 嵌入式软件需要一个较好的操作系统开发平台,提供性能完备的实时控制、任务管理、存储管理和资源分配等功能。在这个平台上进行应用软件的开发,程序员不必去考虑底层的实现细节。
- 嵌入式系统开发工具要求易学、易用、可靠、高效、人机用户界面友好,为程序员提供一个方便好用的开发环境。
- 支持嵌入式系统开发可剪裁的要求。针对不同的嵌入式系统应用,可以根据需要来剪裁出一个大小适宜的系统。

在实际的嵌入式开发中,根据项目的需要,既可以采用一组相互独立的软件开发工具,

如编辑器、编译器、调试器和仿真器等，也可以采用一些商业化的集成开发环境，将各种软件开发工具集成在一个用户界面友好、功能强大、使用方便、适用性广、覆盖产品开发全周期的平台环境中。

### 4.3.1 宿主机、目标机

嵌入式应用开发需要良好的开发环境的支持。在嵌入式系统中，由于目标机的资源有限，不可能在其上建立庞大、复杂的开发环境，因而通常的做法是把开发环境和目标运行环境进行分离。如图 4-19 所示，嵌入式应用软件的开发方式一般是：在宿主机（host）上建立开发环境，进行应用程序编码和交叉编译，然后在宿主机和目标机（target）之间建立连接，将应用程序下载到目标机上进行交叉调试。经过调试和优化，最后将应用程序固化到目标机中实际运行。

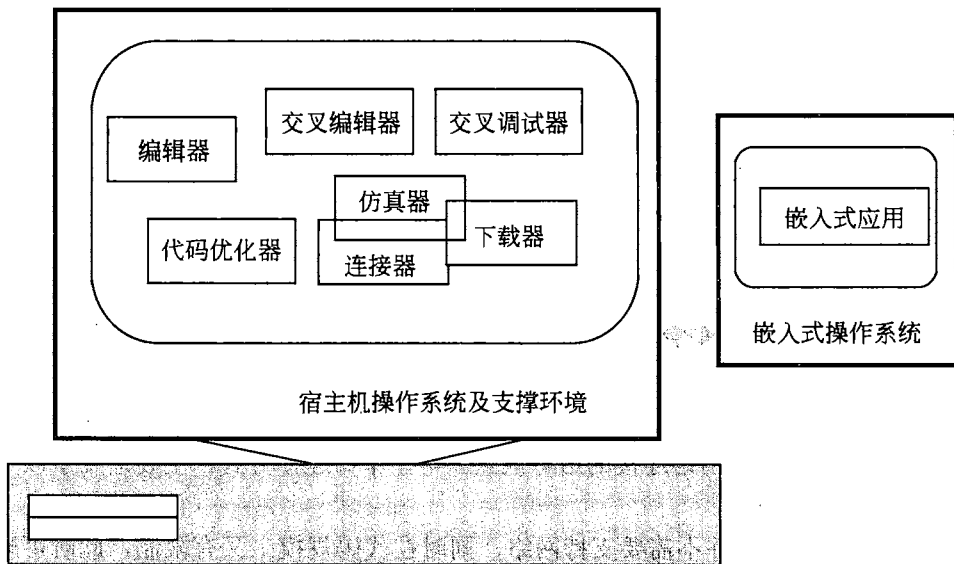


图 4-19 宿主机与目标机的开发模式

#### 1. 宿主机

宿主机是用于开发嵌入式系统的计算机，它通常是拥有大容量内存和硬盘、支持打印机等外设的 PC 机或工作站。在宿主机端（其操作系统可以是 Windows 系列、Linux 或 Solaris 等）运行的工具包括文本编辑器、交叉编译器、交叉调试器、集成环境以及各种分析工具。其中集成环境是其他工具的总入口，被集成的工具一般有它自己独立的图形界面，比如交叉调试器和分析工具等。

## 2. 目标机

目标机一般在嵌入式应用软件的开发和调试期间使用，它可以是嵌入式应用软件的实际运行环境，也可以是能够替代实际运行环境的仿真系统。目标机的软硬件资源通常都有限，主要用来运行包含应用程序代码和嵌入式操作系统的可执行映像。

在开发过程中，目标机端需接收和执行宿主机发出的各种命令，如设置断点、读内存和写内存等，并将结果返回给宿主机，配合宿主机各方面的工作。所有需要与目标机进行信息交互的工具在目标机端都有自己的代理，有的代理是软件实现的（如目标机监控器），有的代理是硬件实现的（如 BDM、JTAG 等）。在目标机端运行的这些代理，负责解释并执行从宿主机端发送过来的各种命令。

## 3. 宿主机与目标机的连接

在宿主机和目标机之间必须建立连接，这样就可以从宿主机向目标机下载、运行可执行映像，或者进行远程调试。宿主机和目标机之间的连接可以分为两类：物理连接和逻辑连接。

物理连接是指宿主机与目标机上的一定物理端口通过物理线路连接在一起。其连接方式主要有三种：串口、以太网接口和 OCD（On Chip Debug）方式（如 JTAG、BDM）等。物理连接是逻辑连接的基础。

逻辑连接是指宿主机与目标机之间按某种通信协议建立起来的通信连接，目前已形成了一些通信协议的标准。

要顺利地建立起交叉开发环境，需要正确地设置这两种连接，缺一不可。在物理连接上，要注意使硬件线路正确连接，且硬件设备完好，能正常工作，连接线路的质量要好；在逻辑连接上，要正确配置宿主机和目标机的物理端口参数，并与实际的物理连接一致。

在实际的嵌入式开发中，最常用的连接方式是以太网上的 IP 网络连接，这种连接不但有很高的带宽，而且具有网络连接的所有优点。至于串口连接方式，主要适用于以下两种情况：

- 在嵌入式应用中并不需要支持网络，同时在代码规模上又有限制，此时可删除嵌入式操作系统中的网络部分。
- 进行嵌入式操作系统内核调试，而有些嵌入式操作系统的网络驱动程序并不支持这种调试模式。

实际上，这两种连接方式是可以并存的。例如，在下载可执行映像时可以使用以太网接口，在进行操作系统内核调试时可以使用串口。

## 4.3.2 嵌入式软件开发工具

嵌入式软件的开发可以分为几个阶段：源代码程序的编写，将源程序交叉编译成各个

目标模块，将所有目标模块及相关的库文件链接成目标程序，代码调试等。在不同的阶段需要用到不同的软件开发工具，如编辑器、编译器、调试工具、软件工程工具等。

### 1. 编辑器

从理论上来说，任何一个文本编辑器都可以用来编写源代码。但是为了提高编程的效率，一个好的编辑器应该具备如下一些特点。

- 支持 C、汇编等程序设计语言的语法高亮显示。
- 支持文件管理操作（如打开文件、保存文件、关闭文件等）、文件编辑操作、文件打印、文本查找等功能。
- 编辑窗口可以同时作为调试时源代码执行的跟踪窗口。
- 通过编译结果输出窗口可以直接定位到相应的源代码编辑窗口。
- 提供一系列辅助编辑工具。
- 编辑器可以同时打开多个窗口进行编辑，可编辑的文件大小理论上无限制。
- 编辑器的编辑命令和编辑操作最好同标准的 Windows 编辑器功能一致，以便熟悉 Windows 操作系统的用户使用。

在各种集成开发环境中，一般都会提供一个功能强大的编辑器。以下介绍的是两个比较好的独立编辑器。

UltraEdit 是一个功能强大的文本编辑器，它可以取代记事本，用来编辑文本文字，也可以用来编写各种语言的源代码。它内建英文单词检查、C++ 及 Visual Basic 语法加亮显示，并可同时编辑多个文件，即使打开一个很大的文件，速度也不会慢。UltraEdit 附有 HTML Tag 颜色显示、搜寻替换以及无限制的还原功能。它支持二进制和十六进制编辑，可以用来直接修改 EXE 或 DLL 文件。

Source Insight 是一款面向工程项目的源码编辑和查看软件，其用户界面友好，变量和函数名都以特定的颜色表示出来，非常直观。对于各种语言的源文件，如 C/C++、C# 和 Java，它能自动解析程序的语法结构，动态地保持符号信息数据库，并主动显示有用的上下文信息。Source Insight 不仅是一个功能强大的程序编辑器，它还能显示参考树、类继承图和调用树等信息。它具有快速源代码导航功能，用户可以使用各种搜索命令，在各个源文件的不同函数和变量定义之间来回跳转，非常方便，所以它很适合于编辑大型软件。

### 2. 编译器

编译阶段要做的工作是用交叉编译或汇编工具处理源代码，产生目标文件。在嵌入式系统中，宿主机和目标机所采用的处理器芯片通常是不一样的。例如，目标机采用的 CPU 是 DragonBall M68x 系列或 ARM 系列，而宿主机采用的是 x86 系列。因此，为了把宿主机上编写的高级语言程序编译成可以在目标机上运行的二进制代码，就需要用到交叉编译器。

与普通 PC 机中的 C 语言编译器不同, 嵌入式系统中的 C 语言编译器要进行专门的优化, 以提高编译效率。一般来说, 优秀的嵌入式 C 编译器所生成的代码, 其长度和执行时间仅比用汇编语言编写的代码长 5%~20%。编译质量的不同, 是区别嵌入式 C 编译器工具的重要指标。因此, 硬件厂商往往会针对自己开发的处理器的特性来定制编译器, 既提供对高级语言的支持, 又能很好地对目标代码进行优化。

GNU C/C++ (gcc) 是目前比较常用的一种交叉编译器, 它支持非常多的宿主机/目标机组合。宿主机可以是 UNIX、AIX、Solaris、Windows、Linux 等操作系统, 目标机可以是 x86、PowerPC、MIPS、SPARC、Motorola 68K 等各种类型的处理器。

gcc 是一个功能强大的工具集合, 包含了预处理器、编译器、汇编器、连接器等组件。它在需要时会去调用这些组件来完成编译任务, 而输入文件的类型和传递给 gcc 的参数决定了它将调用哪一些组件。对于一般或初级的开发者, 它可以提供简单的使用方式, 即只给它提供 C 源码文件, 它将完成预处理、编译、汇编、连接等所有工作, 最后生成一个可执行文件。而对于中高级开发者, 它提供了足够多的参数, 可以让开发者全面控制代码的生成, 这对于嵌入式系统软件开发来说是非常重要的。

gcc 识别的文件类型主要包括: C 语言文件、C++语言文件、预处理后的 C 文件、预处理后的 C++文件、汇编语言文件、目标文件、静态链接库、动态链接库等。以 C 程序为例, gcc 的编译过程主要分为 4 个阶段。

(1) 预处理阶段, 即完成宏定义和 include 文件展开等工作。

(2) 根据编译参数进行不同程度的优化, 编译成汇编代码。

(3) 用汇编器把上一阶段生成的汇编码进一步生成目标代码。

(4) 用连接器把上一阶段生成的目标代码、其他一些相关的系统目标代码以及系统的库函数连接起来, 生成最终的可执行代码。

用户可以通过设定不同的编译参数, 让 gcc 在编译的不同阶段停止下来, 这样可以检查编译器在不同阶段的输出结果。

在 gcc 的高级用法上, 一般希望通过使用编译器达到两个目的: 检查出源程序的错误; 生成速度快、代码量小的执行程序。这可以通过设置不同的参数来实现, 例如, “-Wall” 参数可以发现源程序中隐藏的错误; “-O2” 参数可以优化程序的执行速度和代码大小; “-g” 参数可以对执行程序进行调试。

### 3. 调试及调试工具

在开发嵌入式软件时, 交叉调试是必不可少的一步。嵌入式软件的特点决定了其调试具有如下特点。

- 对于通用的计算机, 调试器 (debugger) 与被调试程序 (debugged) 一般位于同一台计算机上, 操作系统也相同, 调试器进程通过操作系统提供的调用接口来控制

被调试的进程。而在嵌入式系统中，由于目标机的资源有限，调试器和被调试程序运行在不同的机器上，调试器主要运行在宿主机上，而被调试程序则运行在目标机上。

- 调试器通过某种通信方式与目标机建立联系。通信方式可以是串口、并口、网络、JTAG 或专用的通信方式。
- 在目标机上一般有调试器的某种代理 (agent)，这种代理能配合调试器一起完成对目标机上运行的程序的调试。这种代理可以是某种软件，也可以是支持调试的某种硬件。

总之，在交叉调试方式下，调试器和被调试程序运行在不同的机器上。调试器通过某种方式能控制目标机上被调试程序的运行方式，并能查看和修改目标机上的内存、寄存器以及被调试程序中的变量。在嵌入式软件的开发实践中，经常采用的调试方法有直接测试法、调试监控器法、ROM 仿真器法、在线仿真器法、片上调试法及模拟器法。

#### (1) 直接测试法

直接测试法是嵌入式系统发展早期经常采用的一种调试方法。这种方法需要的调试工具非常简单，比较适合当时的实际情况。采用这种方式进行软件开发的基本步骤是：

- ① 在宿主机上编写程序的源代码。
- ② 在宿主机上反复地检查源代码，直到编译通过，生成可执行程序。
- ③ 将可执行程序固化到目标机上的非易失性存储器（如 EPROM、Flash 等）中。
- ④ 在目标机上启动程序运行，并观察程序的运行结果。
- ⑤ 如果程序不能正常工作，则在宿主机上反复检查代码，查找问题的根源。然后修改代码，纠正错误，并重新编译。
- ⑥ 重复步骤③~⑤，直到程序能正常工作。

从这些开发步骤可以看出，这种调试方法基本上无法监测程序的运行。虽然也有人提出了一些调试的小窍门，例如，从目标机打印一些有用的提示信息（通过监视器、LCD 或串口等输出信息），或者利用目标机上的 LED 指示灯来判断程序的运行状态。但这些窍门的作用有限，如果一个程序在运行时没有产生预想的效果，那么开发者只能通过检查源程序来发现问题。显然，这种调试方法的效率很低，难度很大，开发人员也很辛苦。但由于开发条件，特别是开发工具的限制，在嵌入式系统的早期阶段，程序的开发只能采用这种方法，甚至目前在开发一些新的嵌入式产品时，也往往要采用这种方法。

#### (2) 调试监控器法

调试监控器法的工作原理如图 4-20 所示。在这种调试方式下，调试环境由三部分构成，即宿主机端的调试器、目标机端的监控器（监控程序）以及两者之间的连接（包括物理连接和逻辑连接）。



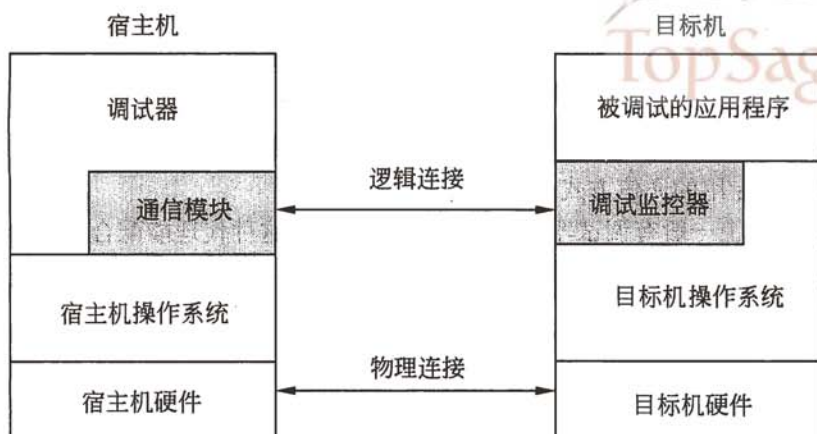


图 4-20 调试监控器法的工作原理

监控器是运行在目标机上的一段程序，它负责监视和控制目标机上被调试程序的运行，并与宿主机端的调试器一起，完成对应用程序的调试。监控器预先被固化到目标机的 ROM 空间中，在目标机复位后将被首先执行。它对目标机进行一些必要的初始化，然后初始化自己的程序空间，最后等待宿主机端的命令。监控器能配合调试器完成被调程序的下载、目标机内存和寄存器的读/写、设置断点以及单步执行被调试程序等功能。一些高级的监控器具有能配合完成代码分析 (code profiling)、系统分析 (system profiling)、ROM 空间的写操作等功能。

利用监控器方式作为调试手段时，开发应用程序的步骤如下：

- ① 启动目标机，监控器掌握对目标机的控制，等待与调试器建立连接。
- ② 调试器启动，与监控器建立起通信连接。
- ③ 调试器将应用程序下载到目标机上的 RAM 空间中。
- ④ 开发人员使用调试器进行调试，发出各种调试命令。监控器解释并执行这些命令，并通过目标机上的各种异常来获得对目标机的控制，将命令执行结果回传给调试器。
- ⑤ 如果程序有问题，则开发人员在调试器的帮助下定位错误，修改之后再重新编译连接并下载程序，开始新的调试。如此反复直到程序能正确运行为止。

监控器方式明显地提高了程序调试的效率，降低了调试的难度，缩短了产品的开发周期，有效地降低了开发成本。而且这种方法的成本也比较低廉，基本上不需要专门的调试硬件支持。因此，它是目前使用最为广泛的嵌入式软件调试方式之一，几乎所有的交叉调试器都支持这种方式。

### (3) ROM 仿真器法

ROM 仿真器可以认为是一种用于替代目标机上 ROM 芯片的硬件设备。它一边同宿主主机相连，一边通过 ROM 芯片的插座同目标机相连。对于嵌入式处理器，它就像一个只读存储芯片；而对于宿主机上的调试器，它又像一个调试监控器。由于仿真器上的地址可以实时地映射到目标机的 ROM 地址空间中，所以在目标机上可以没有 ROM 芯片，而是用仿真器提供的 ROM 空间来代替。

实际上 ROM 仿真器是一种不完全的调试方式，它只是为目标机提供 ROM 芯片，并在目标机和宿主机之间建立了一条高速的通信通道。因此它经常同调试监控器法相结合，形成一种功能更强的调试方法。

与简单的监控器方法相比，ROM 仿真器的优点是：

- 在目标机上可以没有 ROM 芯片，所以也就不需要用其他的工具来向 ROM 中写入数据和程序。
- 省去了为目标机开发调试监控器的麻烦。
- 由于是通过 ROM 仿真器上的串行接口、并行接口或网络接口与宿主机相连，所以不必占用目标机上通常很有限的资源。

#### (4) 在线仿真器法

在线仿真器(In Circuit Emulator, ICE)是一种用于替代目标机上 CPU 的设备。对目标机来说，在线仿真器就相当于它的 CPU。事实上，ICE 本身就是一个嵌入式系统，有自己的 CPU、RAM、ROM 和软件。它的 CPU 比较特殊，可以执行目标机 CPU 的所有指令，但有更多的引出线，能将内部信号输出到被控制的目标机上。在线仿真器的存储器也可以被映射到用户的程序空间上。因此，即使没有目标机，仅用 ICE 也可以进行程序的调试。

ICE 和宿主机一般通过串口、并口或网络相连。在连接 ICE 和目标机时，需要先将目标机的 CPU 取下，然后将 ICE 的 CPU 引出线接到目标机的 CPU 插槽上。在使用 ICE 来调试程序时，在宿主机上也有一个调试器用户界面，在调试过程中，这个调试器将通过 ICE 来控制目标机上的程序。

采用在线仿真器，可以完成如下的调试功能：

- 同时支持软件断点和硬件断点的设置。软件断点只能到指令级别，也就是说，只能指定程序在取某一指令前停止运行。而在硬件断点方式下，多种事件的发生都可使程序在一个硬件断点上停止运行，这些事件不仅包括取指令，还包括内存读/写、I/O 读/写以及中断等。
- 能够设置各种复杂的断点和触发器。例如，可以让程序在“当变量 me 等于 100，同时 AX 寄存器等于 0”时停止运行。
- 能实时跟踪目标程序的运行，并可实现选择性的跟踪。在 ICE 上有大块 RAM，专门用来存储执行过的每个指令周期的信息，使用户可以得知各个事件发生的精确



次序。

- 能在不中断被调试程序运行的情况下查看内存和变量，即非干扰的调试查询。

在线仿真器特别适用于调试实时应用系统、设备驱动程序以及对硬件进行功能测试。它的主要缺点就是价格太昂贵，一般都在几千美元，有的甚至要几万美元。这显然阻碍了团队的整体开发，因为不可能给每位开发人员都配备一套在线仿真器。因此，现在 ICE 一般都用于普通调试工具解决不了的问题，或者用它来做严格的实时性能分析。

### (5) 片上调试法

片上调试 (On Chip Debugging, OCD) 是 CPU 芯片提供的一种调试功能，可以把它看成是一种廉价的 ICE 功能。OCD 的价格只有 ICE 的 20%，但却提供了 80% 的 ICE 功能。

最初的 OCD 是一种仿调试监控器方式，即将监控器的功能以微码的形式来体现，如 Motorola 的 CPU 32 系列处理器。后来的 OCD 摒弃了这种结构，采用了两级模式的思路，即将 CPU 的工作模式分为正常模式和调试模式。

当满足了特定的触发条件时，CPU 就可进入调试模式。在调试模式下，CPU 不再从内存读取指令，而是从调试端口读取指令，通过调试端口可以控制 CPU 进入和退出调试模式。这样在宿主机端的调试器就可以直接向目标机发送要执行的指令，通过这种形式调试器可以读/写目标机的内存和各种寄存器，控制目标程序的运行以及完成各种复杂的调试功能。

OCD 方式的主要优点是：不占用目标机上的通信端口等资源；调试环境和最终的程序运行环境基本一致；支持软硬件断点；提供跟踪功能，可以精确计量程序的执行时间；支持时序分析等功能。

OCD 方式的主要缺点是：调试的实时性不如 ICE 强；不支持非干扰的调试查询；使用范围受限，目标机上的 CPU 必须具有 OCD 功能。

目前比较常用的 OCD 的实现有：后台调试模式 (Background Debugging Mode, BDM)、连接测试存取组 (Joint Test Access Group, JTAG) 和片上仿真器 (On Chip Emulation, OnCE) 等。其中，JTAG 是主流的 OCD 方式，而 OnCE 是 BDM 和 JTAG 的一种融合方式。

### (6) 模拟器法

模拟器是一个运行在宿主机上的纯软件工具，它通过模拟目标机的指令系统或目标机操作系统的系统调用来达到在宿主机上运行和调试嵌入式程序的目的。

模拟器主要有两种类型：一类是在宿主机上模拟目标机的指令系统，称为指令级的模拟器；另一类是在宿主机上模拟目标机操作系统的系统调用，称为系统调用级的模拟器。指令级模拟器相当于在宿主机上建立了一台虚拟的目标机，该目标机的 CPU 种类与宿主机不同。例如，宿主机的 CPU 是 Intel Pentium，而虚拟机是 ARM、PowerPC 或 MIPS 等。比较高级的指令级模拟器还可以模拟目标机的外部设备，如键盘、串口、网口和 LCD 等。系统调用级的模拟器相当于在宿主机上安装了目标机的操作系统，使得基于目标机操作系统

的应用程序可以在宿主机上运行。两种类型的模拟器相比,指令级模拟器所提供的运行环境与实际的目标机更接近;而系统调用级的模拟器本身比较容易开发,也容易移植。

使用模拟器的最大好处是:可以在实际的目标机环境并不存在的条件下开发其应用程序,并且在调试时可以利用宿主机的资源来提供更详细的错误诊断信息。但模拟器也有许多不足之处,包括:

- 模拟环境与实际的运行环境差别较大,无法保证在模拟条件下调试通过的程序就一定能在真实环境下顺利运行。
- 不能模拟所有的设备。嵌入式系统中经常包含许多外围设备,但除了一些比较常见的设备之外,多数设备是不能模拟的。
- 实时性差。在使用模拟器调试程序时,被调试程序的执行时间同在真实环境中的运行时间差别较大。

尽管模拟器有许多不足,但是在项目开发的早期阶段,尤其是在还没有任何硬件可供使用时,它还是非常有用的。对那些实时性不强,没有特殊外设,只需验证其逻辑的程序,用模拟器基本可以完成所有的调试工作。而且在使用模拟器调试程序时,不需要额外的硬件来协助,所以降低了开发成本。

#### 4. 软件工程工具

软件工程工具是指在分布式开发环境或大型嵌入式软件项目中使用的各种管理软件,如 CVS、make 等。

##### (1) CVS

CVS (Concurrent Version System) 是一个版本控制软件,用来记录源码文件和其他相关文件的修改历史。对于一个文件的各个版本,CVS 只存储版本之间的区别,而不是把每个版本都完整地保存下来。当一个文件的内容发生变化时,CVS 会在一个日志中记录每一次修改的作者、修改的时间以及修改的原因。CVS 能够有效地管理软件的发行版本,以及多位程序员同时参与的分布式开发环境。它把一个软件项目组织成一个层次化的目录结构,里面包含了与项目有关的所有文件,如源文件、文档文件等。这些目录和文件合并起来,就构成了该软件项目的一个发行版本。

##### (2) GNU make

GNU make 是一种代码维护工具,在大中型软件开发项目中,它将根据程序各个模块的更新情况,自动地维护和生成目标代码。make 的主要任务是读入一个文本文件(默认的文件名是 makefile 或 Makefile),并根据这个文件所定义的规则和步骤,完成整个软件项目的维护和代码生成等工作。在这个文本文件中,定义了一些依赖关系(即哪些文件的最新版本是依赖于哪些其他的文件)和需要什么命令来产生文件的最新版本或管理各种文件。有了这些信息,make 会检查文件的修改或生成时间戳,如果目标文件的时间戳比它的某个

依赖文件要旧,那么 **make** 就会执行 **makefile** 文件中描述的相应命令来更新目标文件。**make** 工具的特点如下:

- 适合于文件较多的大中型软件项目的编译、连接、清除中间文件等管理工作。
- 只更新那些需要更新的文件,而不重新处理那些并不过时的文件。
- 提供和识别多种默认规则,方便对大型软件项目的管理。
- 支持对层状目录结构的软件项目进行递归管理。
- 对软件项目,具有渐进式的可维护性和扩展性。

### 4.3.3 集成开发环境

嵌入式软件开发环境起初主要由专门开发工具的公司提供,这些公司根据不同操作系统和不同处理器版本进行专门定制,如美国 Microtec 公司的交叉开发工具曾经被 VRTX、pSOS 等定制采用。随着用户对开发工具套件的需求增加,一些著名的操作系统供应商开始发展本系列操作系统产品的开发工具套件,如 WindRiver 公司的 Tornado、微软的 Windows CE 嵌入式开发工具包等。

在国际上,嵌入式软件开发环境的另一支研发队伍是 GNU。他们在因特网上提供免费的相关研究和开发成果,成为自主开发嵌入式软件开发环境的重要资源。一些公司已在 GNU 软件的基础上,经过集成、优化和测试,推出更加成熟、稳定的商业化嵌入式软件开发环境。

随着嵌入式系统的发展,嵌入式软件开发环境越来越重要,它直接影响到嵌入式软件的开发效率和质量。目前的开发环境已向开放性、集成化、可视化和智能化的方向发展,将各种类型的功能强大的软件工具,如编辑器、编译器、连接器、调试器、版本管理、用户界面等有机地集成在一个统一的集成开发环境(Integrated Development Environment, IDE)中。

#### 1. Tornado

Tornado 是 WindRiver 公司推出的一个集成开发环境。它由三个高度集成的部分组成:运行在宿主机和目标机上的交叉开发工具和实用程序;运行在目标机上的实时操作系统 VxWorks;用来连接宿主机和目标机的各种通信介质,如以太网、串口、在线仿真器 ICE 或 ROM 仿真器等。

Tornado 提供的交叉开发工具和实用工具主要有:源代码编辑工具、图形化的交叉调试工具、工程配置工具、集成仿真工具、诊断分析工具、C/C++编译工具、宿主机-目标机连接配置工具、目标机系统状态浏览工具、命令行执行工具、多语言浏览工具及图形化内核配置工具等。在 Tornado 中,宿主机上的工具与目标机之间的通信由目标服务器和目标代理共同完成。如图 4-21 所示,在形式上目标代理是 VxWorks 上的一个任务,调试命令

通过宿主机上的目标服务器发送给目标代理。这些调试请求决定了目标代理应如何控制目标机上的其他任务。

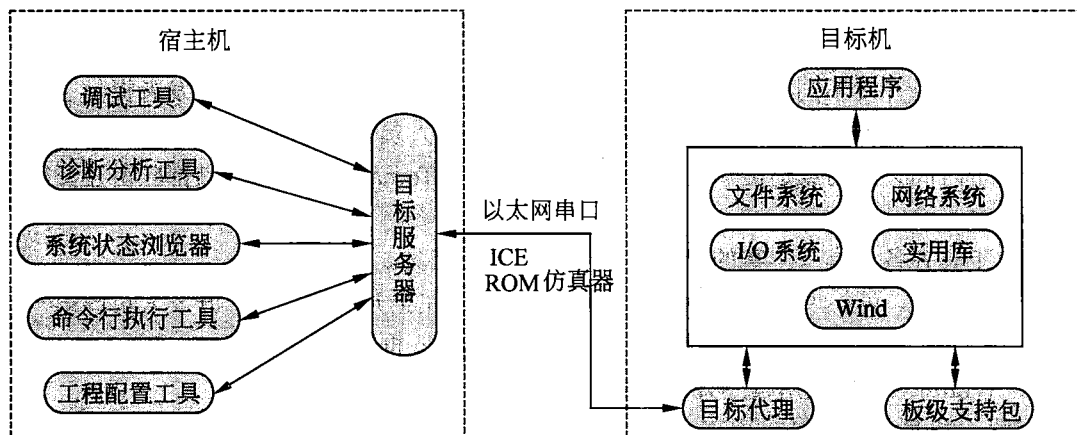


图 4-21 Tornado 环境中宿主机与目标机之间的关系

- 图形化的交叉调试器 CrossWind/WDB: 支持任务级和系统级两种调试方式、支持混合源代码和汇编代码显示、支持多目标同时调试, 具有良好的图形用户界面。
- 工程配置工具 Project: 用于对 VxWorks 操作系统及其组件进行自动配置, 进行依赖性分析和代码容量计算, 自动生成 Makefile 文件。
- 集成仿真工具 VxSim: 提供与真实目标机完全一致的调试和仿真运行环境。
- 诊断分析工具 WindView: 一个图形化的动态诊断和分析工具, 主要是向开发者提供在目标机上运行的应用程序的许多详细情况。
- C/C++编译工具: Tornado 提供支持 C/C++语言的工具和类库——Diab C/C++编译器、GNU C/C++编译器及 iostreams 类库。
- 宿主机-目标机连接配置工具 Launcher: 位于 Tornado 环境的最上层, 开发者可以通过它来设置开发环境。
- 目标机系统状态浏览工具 Browser: 一个图形化工具, 能随时提供目标系统的全面状态信息。
- 命令行执行工具 WindSh: 一个功能强大的命令行解释器, 可以直接解释、执行 C 语言表达式, 调用目标机上的 C 函数及访问已在系统符号表中定义的变量。
- 多语言浏览工具 WindNavigator: 浏览源程序代码, 用图形化的方式显示函数调用关系, 从而实现快速的代码定位。
- 图形化内核配置工具 WindConfig: 通过 WindConfig 提供的图形向导, 用户可以方

便地配置 VxWorks 内核及其组件的参数。

Tornado 的特点:

- 友好的开发环境。Tornado 可以运行在不同的系统中, 支持 UNIX、Windows NT、Windows 95/98 等。
- 适用于开发不同类型的目标机。针对不同的目标机, Tornado 为开发者提供了一个一致的图形接口和人机界面。这样, 当开发人员转向新的目标机时, 不必再花费时间去学习或适应新的开发工具。事实上, Tornado 的所有工具都驻留在开发平台上。
- 工具齐备, 具有丰富的交叉开发工具和实用工具。
- 开放的、可扩展的开发环境。Tornado 是一个完全开放的环境, 开发人员或第三方厂商可以很容易地把自己的工具集成到 Tornado 框架下。

## 2. Windows CE 应用程序开发工具

如图 4-22 所示, Windows CE 应用程序开发工具包括: Platform Builder、eMbedded Visual Tools、eMbedded Visual Basic、eMbedded Visual C++。它们都是专门针对 Windows CE 操作系统的开发工具。

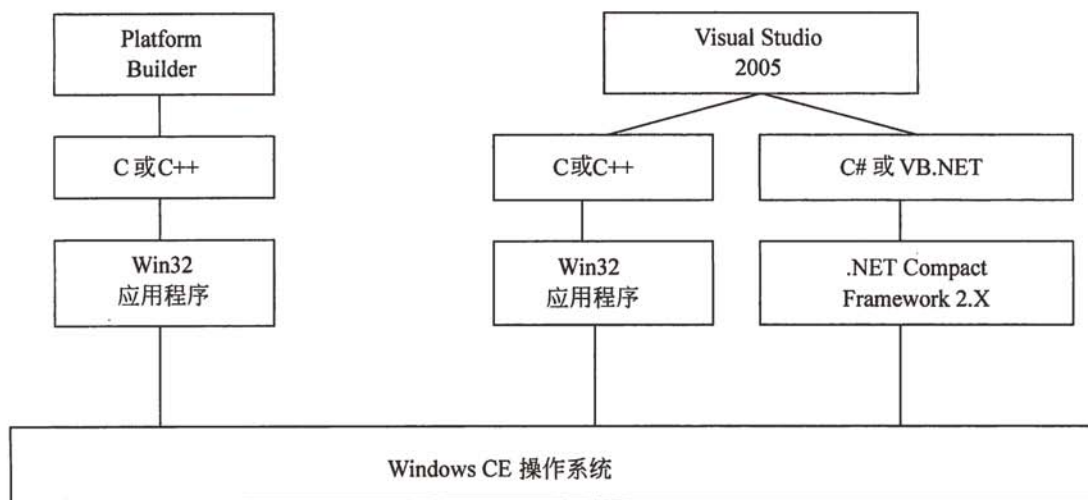


图 4-22 Windows CE 应用程序开发工具

Microsoft Windows CE Platform Builder 为开发商迅速创建一个嵌入式系统提供了全部相关的工具。Platform Builder 集成开发环境使开发商能够对新一代高度模块化的设计进行配置、创建与调试, 以实现嵌入式系统的灵活性与可靠性同 Windows 和 Web 功能特性之间的紧密结合。它的特点主要包括:

- 通过使用改进的目标-宿主集成与连接特性来提高工作效率,节省嵌入式系统的创建时间。这些特性包括:集成化连接与下载、集成化目标控制、状态监视器、灵活的创建选择、简化操作系统配置等。
- 通过使用先进的系统级调试功能来提高调试速度。Platform Builder 的系统级调试器目前可为硬件辅助和系统级调试提供支持,从而大大扩展了调试功能的作用范围。具体包括:硬件辅助调试、源点级调试、新型的内核追踪器、远程系统信息、远程性能监视器、改进的调试器用户界面、调试区间等。
- 提供了一个新型扩展模型,可以帮助开发商将各类特性集成到开发环境当中,包括微处理器控制单元、嵌入式开发工具控制单元等。

Microsoft eMbedded Visual C++和 eMbedded Visual Basic 是开发下一代 Windows CE 通信、娱乐及信息访问等应用程序时的功能强大的工具。它们所提供的全面高速应用程序开发环境能够帮助开发商在各类不同设备上迅速对相关的 Windows CE 应用程序进行创建、调试和部署,并且在不牺牲控制功能、性能表现及有关灵活性的前提下,提高 Windows CE 的开发效率。这两个产品的特点主要包括以下几条。

- 开发商的工作效率比较高:这两个集成开发环境与传统的 Windows 应用开发环境非常相似,所以开发人员无需额外的培训即可熟练掌握它们的使用方法。而编程时的在线提示辅助功能(如语句完成、参数信息和语法错误检查等),能够大大提高程序员的工作效率。另外,通过创建可重复使用的 ActiveX 组件,可以将软件开发的复杂程度降至最低水平。
- 开发过程与集成化调试得以简化:允许 eMbedded Visual Tools 在编译完成后,从移动设备或模拟器上的 IDE 中自动复制并启动相关的应用程序,从而实现应用程序的迅速测试和执行。在调试程序时,可以使用集成化调试器,当应用程序在 Windows CE 设备上(或模拟器内)运行的同时对其错误予以消除。另外,在测试时可以先在 Windows CE 设备模拟器上对应用程序进行测试,以避免高昂的硬件成本投资。
- 针对 Windows CE 平台的全面访问:包括 TCP/IP 通信机制、COM 组件模型、ActiveX 控件、设备的 API 接口函数等。
- 面向最新的 Windows CE 设备创建相应的解决方案:例如,Handheld PC Pro、Palm-size PC 及 Pocket PC 等 Windows CE 设备。
- 迅速、灵活的数据访问:数据存储可通过相关连接来实现与远程数据源之间的同步。

### 3. Linux 环境下的集成开发环境

在 Linux 环境下也有一些很好的集成开发环境,如 Kdevelop、Eclipse 和 Anjuta 等。

#### (1) Kdevelop

Kdevelop 是 KDE 小组开发的 Linux/UNIX 操作系统上的 C/C++ 集成开发环境, 为快速开发 C/C++ 应用程序提供了强有力的开发工具。

Kdevelop 的操作界面类似于微软公司的 Visual Studio, 它提供了编辑、编译、连接、除错、版本管理及计划管理等基本的 IDE 功能; 它还内建了一个可以产生 Qt 图形界面的资源编辑程序; 对于 C++ 程序, 它则额外提供了类浏览器; 其文件管理程序内建了所有有关 KDE 发展所需的文件, 并提供搜寻的功能。

### (2) Eclipse

Eclipse 是替代 IBM Visual Age for Java (简称 IVJ) 的下一代 IDE 开发环境, 但它未来的目标不仅仅是成为专门开发 Java 程序的 IDE 环境。根据 Eclipse 的体系结构, 通过开发插件, 它能扩展到任何语言的开发, 甚至能成为图形绘制的工具。目前, Eclipse 已经开始提供 C 语言开发的功能插件。Eclipse 是一个开放源代码的项目, 任何人都可以下载其源代码, 并在此基础上开发自己的功能插件。

### (3) Anjuta

Anjuta 是 GNU/Linux 平台下的 C/C++ 集成开发环境, 它主要是为了开发 GTK/GNOME 程序而设计的。Anjuta 利用 GLADE 来生成优美的用户界面, 加之以自己强大的源程序编辑功能, 使之成为一个极好的应用程序快速开发的集成开发环境。以前, 人们使用 GLADE 做界面, 用 emacs 或 vi 来编辑源程序, 再用某种终端模拟器编辑开发项目, 而现在使用 Anjuta, 所有这些繁杂零散的工作都可以在一个统一的、集成的环境下完成。

## 4.4 嵌入式软件开发

### 4.4.1 嵌入式平台选型

按照常规的工程设计方法, 嵌入式系统的设计可以分为三个阶段: 分析、设计和实现。分析阶段是确定要解决的问题及需要完成的目标, 也常常被称为需求阶段; 设计阶段主要是解决如何在给定的约束条件下完成用户的要求; 实现阶段主要是解决如何在所选择的硬件和软件的基础上进行整个软、硬件系统的协调和实现。在分析阶段结束后, 开发者通常面临的一个棘手问题就是软硬件平台的选择, 因为它的好坏直接影响着实现阶段的任务完成。

通常, 硬件和软件的选择包括处理器、硬件部件、操作系统、编程语言、软件开发工具、硬件调试工具和软件组件等。

#### 1. 硬件平台的选择

嵌入式系统的核心部件是各种类型的嵌入式处理器。据不完全统计, 目前全世界嵌入式处理器的品种总量已经超过一千多种, 流行的体系结构有三十多个系列。由于嵌入式系

统设计的差异极大，所以选择是多样化的。

设计者在选择处理器时要考虑的因素主要有以下几个方面。

- 处理性能：对于许多需使用处理器的嵌入式系统设计来说，目标不是在于挑选速度最快的处理器，而是在于选取能够完成操作的处理器和 I/O 子系统。
- 技术指标：当前许多嵌入式处理器都集成了外围设备的功能，减少了芯片的数量，降低了整个系统的开发费用。开发人员首先考虑的是系统所要求的一些硬件能否无须过多的逻辑就连接到处理器上，其次考虑该处理器的一些支持芯片的配套情况。
- 功耗：对于手持设备、PDA、手机等消费类电子产品，在选购微处理器时要求高性能、低功耗。
- 软件支持工具：选择合适的软件开发工具对系统的实现会起到很好的作用。
- 是否内置调试工具：处理器如果内置调试工具，可以大大缩短调试周期，降低调试难度。

## 2. 软件平台的选择

软件平台的选择涉及到操作系统、编程语言和集成开发环境三个方面。

### (1) 操作系统

对于编写嵌入式软件，有两种选择：一是自己编写内核；二是使用现成的操作系统。如果嵌入式软件只需要完成一项非常小的工作，比如在电动玩具、空调中，就不需要一个功能完整的操作系统。但如果系统的规模较大、功能较复杂，那么最好还是使用一个现成的操作系统。可用于嵌入式系统软件开发的操作系统有很多，但关键是如何选择一个适合开发项目的操作系统，这可以从以下几点进行考虑。

- 操作系统提供的开发工具：有些实时操作系统只支持该系统供应商的开发工具，因此，还必须从操作系统供应商处获得编译器、调试器等；而有的操作系统应用广泛，且有第三方工具可用，所以选择的余地比较大。
- 操作系统向硬件接口移植的难度：操作系统向硬件的移植是一个重要的问题，是关系到整个系统能否按期完工的一个关键因素。因此，要选择那些可移植性程度高的操作系统，以避免因移植带来的种种困难。
- 操作系统的内存要求，有些操作系统对内存有较大要求。
- 操作系统的可剪裁性、实时性能等。

### (2) 编程语言

尽管高级语言能够完成大部分的嵌入式软件开发工作，但汇编语言仍然不可替代。汇编语言可以直接对硬件进行操作，代码效率高，所以经常应用在系统移植以及需要直接控制硬件的场合。此外，良好的汇编基础也有助于程序的调试。



越是高级的语言，其编译和运行的系统开销就越大，应用程序也越大，运行越慢。因此一般来说，编程人员都会首选汇编语言和 C 语言，然后才会考虑 C++ 语言或 Java 语言。

### (3) 集成开发环境

集成开发环境是进行开发时的重要平台，在选择时应考虑以下因素：

- 系统调试器的功能，包括远程调试环境。
- 支持库函数。许多开发系统提供大量的库函数和模板代码，如 C++ 编译器就带有标准的模板库。与选择硬件和操作系统的原则一样，应尽量采用标准的 glibc (GNU 标准 C 库函数)。
- 连接程序是否支持所有的文件格式和符号格式。

## 4.4.2 软件设计

### 1. 软件设计的任务

在给定系统的需求规格说明书后，需要对软件的结构进行设计，并对设计的过程进行管理。在嵌入式系统的软件设计过程中，需要完成以下一些任务。

#### (1) 准备一个工作计划

在软件设计之前，首先要制定一个详细的工作计划，其内容包括下面几条。

- 过程管理方案：包括软件开发的进度管理、软件规模和所需人年的估算、开发人员的技能培训等。
- 开发环境的准备方案：包括开发工具的准备、开发设备的准备、测试装备的准备、分布式开发环境下的开发准则等。
- 软硬件联机调试的方案：联调的起始时间、地点、人员和具体的准备工作。
- 质量保证方案：包括质量目标计划、质量控制计划等。
- 配置控制方案：包括配置控制文档的编写、配置控制规则的制定等。

#### (2) 确定软件的结构

设计软件的各个组成部分，包括以下几点。

- 任务结构的设计：使用操作系统提供的函数，设计出一个最佳的任务结构。
- 线程的设计。
- 公共数据结构的设计：在确保系统一致性的基础上，设计出所需的公共数据。
- 操作系统资源的定义。
- 类的设计。
- 模块结构设计：在设计时要充分考虑模块的划分、标准化、可重用和灵活性等。
- 内存的分配与布局。

#### (3) 设计评审

对于软件设计的结果，进行一次设计评审，并在必要时对设计进行修正。具体内容

包括：

- 确认每件工作的执行方法是否恰当，其内容是否完善。
- 确认该设计完成了系统需求规格说明书所要求的功能和服务。
- 评估任务结构设计、评估类的设计、评估模块结构设计。
- 对软件设计的结果进行总结，编写出相应的文档。

#### (4) 维护工作计划

执行软件设计工作控制，在每周和每月的时间粒度上对进度进行控制，确保软件设计能够如期完成。

#### (5) 与硬件部门密切合作、相互协调

根据工作计划中的安排，定期与硬件部门召开会议，协调各自的进展。如果软件规格说明书发生了变化，立即进行调整，重新进行软件设计。

#### (6) 控制工作的结果，把工作记录存档

掌握当前的工作进展情况，尽早地发现和分析问题，并采取相应的措施。对各种事件进行跟踪记录，包括：

- 执行过程控制，跟踪进展情况并定期记录、存档。
- 执行质量控制，保留质量记录。
- 记录产品的配置、版本变化、缺陷的发现和ae理等信息。

## 2. 模块结构设计

模块结构设计的基本任务是：将系统划分为模块，确定软件的结构，模块的功能和模块间的接口，以及全局数据结构的设计。模块结构设计是软件开发过程中很关键的一步。软件的质量及一些整体特性基本上是这一步决定的。

### (1) 模块的概念

模块是组成系统的基本单位，它的特点是可以组合、分解和更换。系统中任何一个处理功能都可以被看成是一个模块。根据模块功能具体化程度的不同，可以分为逻辑模块和物理模块。在系统逻辑模型中定义的处理功能可视为逻辑模块；物理模块是逻辑模块的具体化，可以是一个计算机程序、子程序或若干条程序语句，也可以是人工过程的某项具体工作。

一个模块应具备以下四个要素。

- 输入和输出：模块的输入来源和输出去向都是同一个调用者，即一个模块从调用者那里取得输入，进行加工后再把输出返回给调用者。
- 处理功能：指模块把输入转换成输出所做的工作。
- 内部数据：指仅供该模块本身引用的数据。

- 程序代码：指用来实现模块功能的程序。

前两个因素是模块的外部特性，即反映了模块的外貌；后两个因素是模块的内部特性。在结构化设计中，主要考虑的是模块的外部特性，其内部特性只做必要了解即可。

## (2) 模块结构图

为了保证系统设计工作的顺利进行，模块结构设计应遵循如下原则。

- 所划分的模块其内部的凝聚性要强，模块之间的联系要少，即模块具有较强的独立性。
- 模块之间的连接只能存在上下级之间的调用关系，不能有同级之间的横向关系。
- 整个系统呈树状结构，不允许网状结构或交叉调用关系出现。
- 所有模块都必须严格地分类编码并建立归档文件。

模块结构图主要关心的是模块的外部属性，即上下级模块、同级模块之间的数据传递和调用关系，而不关心模块的内部。

模块结构图是结构化设计中描述系统结构的图形工具。作为一种文档，它必须严格定义模块的名字、功能和接口，同时还应当在模块结构图上反映出结构化设计的思想。模块结构图由模块、调用、数据、控制和转接等五种基本符号组成，如图 4-23 所示。



图 4-23 模块结构图的基本符号

- 模块：这里所说的模块通常是指用一个名字就可以调用的一段程序语句。在长方形的中间，可以标上能反映模块处理功能的模块名字。
- 调用：箭头总是由调用模块指向被调用模块，但是应该理解为被调用模块执行后又返回到调用模块。如果一个模块是否调用一个从属模块决定于调用模块内部的判断条件，则该调用称为判断调用，采用菱形符号表示；如果一个模块通过其内部的循环的功能来循环调用一个或多个从属模块，则该调用称为循环调用，用弧形箭头表示。判断调用和循环调用的表示方法如图 4-24 所示。

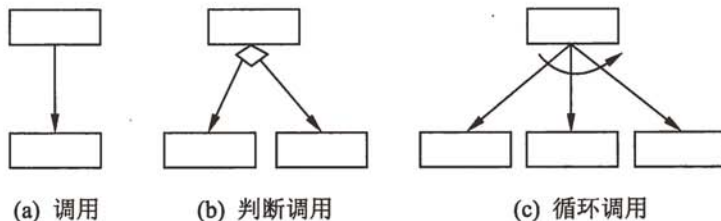


图 4-24 模块调用示例

- 数据：当一个模块调用另一个模块时，调用模块可以把数据传送到被调用模块处供处理，而被调用模块又可以将处理的结果送回到调用模块。在模块之间传送的数据，使用与调用箭头平行的带空心圆的箭头表示，并在旁边标上数据名。
- 控制信息：模块间有时还必须传送某些控制信息。例如，数据输入完成后给出的结束标志、文件读到末尾时所产生的文件结束标志等。控制信息与数据的主要区别是前者只反映数据的某种状态，不必进行处理。
- 转接符号：当模块结构在一张纸上画不下，需要转接到另一张纸上，或者为了避免图上线条交叉时，都可以使用转接符号，在圆圈内加上标号。

### 3. 结构化软件设计方法

结构化设计（Structured Design, SD）方法是一种面向数据流的设计方法，它可以与结构化分析方法衔接。结构化设计方法的基本思想是将系统设计成由相对独立、功能单一的模块组成的结构。

#### 1) 信息流的类型

在需求分析阶段，用结构化分析方法产生了数据流图（Data Flow Diagram, DFD）。所谓数据流图，是一种便于用户理解、分析系统数据流程的图形工具。它摆脱了系统的物理内容，精确地在逻辑上描述系统的功能、输入、输出和设计存储等，是系统逻辑模型的重要组成部分。面向数据流的设计能方便地将 DFD 转换成程序结构图。在 DFD 中，从系统的输入数据流到系统的输出数据流的一连串连续变换形成了一条信息流。DFD 的信息流大体上可以分为两种类型：一种是变换流，另一种是事务流。

- 变换流：信息沿着输入通路进入系统，同时将信息的外部形式转换成内部表示，然后通过变换中心（也称主加工）处理，再沿着输出通路转换成外部形式离开系统。具有这种特性的信息流称为变换流。变换流型的 DFD 可以明显地分成输入、变换（主加工）和输出三大部分。
- 事务流：信息沿着输入通路到达一个事务中心，事务中心根据输入信息（即事务）的类型在若干个动作序列（称为活动流）中选择一个来执行，这种信息流称为事务流。事务流有明显的事务中心，各活动流以事务中心为起点呈辐射状流出。

#### 2) 变换分析

变换分析是从变换流型的 DFD 导出程序结构图。

##### (1) 确定输入流和输出流，孤立出变换中心

人们把 DFD 中系统输入端的数据流称为物理输入，系统输出端的数据流称为物理输出。物理输入通常要经过编辑、格式转换、合法性检查、预处理等辅助性的加工才能成为主加工的真正输入（称它为逻辑输入）。从物理输入端开始，一步步向系统的中间移动，可找到离物理输入端最远，但仍可被看成系统输入的那个数据流，这个数据流就是逻辑输入。

同样，由主加工产生的输出（称为逻辑输出）通常也要经过编辑、格式转换、组成物理块、缓冲处理等辅助加工才能变成物理输出。从物理输出端开始，一步步向系统的中间移动，可找到离物理输出端最远，但仍可被看做系统输出的那个数据流，这个数据流就是逻辑输出。

在 DFD 中，从物理输入到逻辑输入的部分构成系统的输入流，从逻辑输出到物理输出的部分构成系统的输出流，位于输入流和输出流之间的部分就是变换中心。

### （2）第一级分解

第一级分解主要是设计模块结构的顶层和第一层。一个变换流型的 DFD 可映射成如图 4-25 所示的程序结构图。图中顶层模块的功能就是整个系统的功能。输入控制模块用来接收所有的输入数据，变换控制模块用来实现输入到输出的变换，输出控制模块用来产生所有的输出数据。

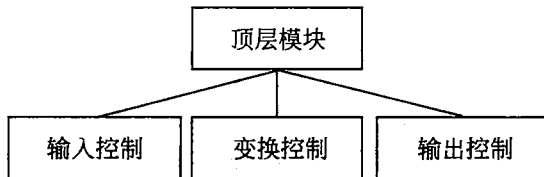


图 4-25 变换分析的第一级分解

### （3）第二级分解

第二级分解主要是设计中、下层模块。

- 输入控制模块的分解：从变换中心的边界开始，沿着每条输入通路，把输入通路上的每个加工映射成输入控制模块的一个低层模块。
- 输出控制模块的分解：从变换中心的边界开始，沿着每条输出通路，把输出通路上的每个加工映射成输出控制模块的一个低层模块。
- 变换控制模块的分解：变换控制模块通常没有通用的分解方法，应根据 DFD 中变换部分的实际情况进行设计。

### （4）事务分析

事务分析是从事务流型 DFD 中导出程序结构图。

① 确定事务中心和每条活动流的流特性。图 4-26 所示给出了事务流型 DFD 的一般形式，其中事务中心（图中的 T）位于数条活动流的起点，这些活动流从该点成辐射状地流出。每条活动流也是一条信息流，它可以是变换流也可以是另一条事务流。一个事务流型的 DFD 由输入流、事务中心和若干条活动流组成。

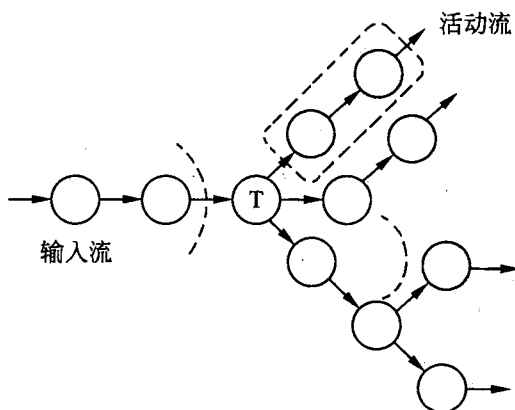


图 4-26 事务流

② 将事务流型 DFD 映射成高层的程序结构。事务流型 DFD 的高层结构如图 4-27 所示。顶层模块的功能就是整个系统的功能；接收模块用来接收输入数据，它对应于输入流；发送模块是一个调度模块，控制下层的所有活动模块；每个活动流模块对应于一条活动流，它也是该活动流映射成的程序结构图中的顶层模块。

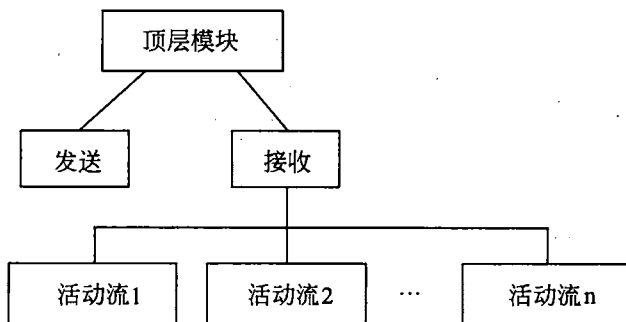


图 4-27 事务流型 DFD 的高层程序结构

③ 进一步分解。接收模块的分解类同于变换分析中输入控制模块的分解。每个活动流模块根据其流特性（变换流或事务流）进一步采用变换分析或事务分析进行分解。

#### (5) 结构化设计方法的设计步骤

- ① 复查并精化数据流图。
- ② 确定 DFD 的信息流类型（变换流或事务流）。
- ③ 根据流类型分别实施变换分析或事务分析。
- ④ 根据系统设计的原则，对程序结构图进行优化。

#### 4. 面向对象软件设计方法

面向对象设计（Object-Oriented Design, OOD）是将面向对象的分析模型转变为软件构造蓝图的设计模型，即在预先定义的基本类框架上构建一个系统。

在 OOD 中，系统由一系列子系统（系统级“模块”）组成，每个子系统又可由下级子系统组成，所有的数据和操作都被封装在“模块”中。OOD 的实现，实际上需要完成一系列不同“模块等级”的设计。

OOD 的独特性在于它基于四个重要的软件设计概念：抽象、信息隐蔽、功能独立和模块性。事实上，现代所有的软件设计方法都试图建造具有这些特征的软件，但只有 OOD 提供了简便而又能达到这一目标的机制。

OOD 分为四个设计层次，从低到高分别是 subsystem 设计层、类及对象设计层、消息设计层和责任设计层。

- 子系统设计层：它包含每个子系统的表示。这些子系统使软件能满足客户定义的需求，并实现支持客户需求的技术基础。该层着重实现主要系统功能的子系统的设计。
- 类和对象设计层：它包含类层次，也包含每个对象的设计。该层使得系统能够以通用化方式创建并不断逼近特殊需求。该层刻画实现系统所需的整体对象体系结构和类层次。
- 消息设计层：它包含每个对象与其他对象/协作者通信的细节，建立系统的内部和外部接口。该层指明如何实现对象间协作。
- 责任设计层：它针对每个对象的所有属性、操作的数据结构和算法的设计。该层标识用于刻画类特征的属性和操作。

### 4.4.3 嵌入式程序设计

#### 1. BootLoader 设计

从操作系统的角度来看，BootLoader 的总目标就是正确地调用内核来执行。

由于 BootLoader 的实现依赖于 CPU 的体系结构，所以大多数 BootLoader 都分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码，例如设备初始化代码等，通常都放在 stage1 中，而且一般都用汇编语言来实现，以达到短小精湛的目的；stage2 则通常用 C 语言来实现，从而可以实现更复杂的功能，而且代码会具有更好的可读性和可移植性。

BootLoader 的 stage1 通常包括以下步骤：

- (1) 硬件设备初始化。
- (2) 为加载 BootLoader 的 stage1 准备内存空间。
- (3) 复制 BootLoader 的 stage1 到内存。

- (4) 设置好堆栈。
- (5) 跳转到 stage1 的 C 入口点。

BootLoader 的 stage2 通常包括以下步骤:

- (1) 初始化本阶段要使用到的硬件设备。
- (2) 检测系统内存映射 (Memory Map)。
- (3) 将内核映像和根文件系统映像从 Flash 上读入到内存当中。
- (4) 为内核设置启动参数, 并调用内核。

## 2. 设备驱动程序设计

下面以嵌入式 Linux 系统为例, 介绍设备驱动程序的设计与实现。

### (1) 概述

大部分的外围物理设备, 如键盘、显示器、鼠标、磁盘、串口、并口、网络适配器等都有一个专用于控制该设备的设备驱动程序。设备驱动是建立在硬件 I/O 设备上的一个抽象层, 这个抽象层的建立可以允许上面的软件层使用统一的、独立于硬件的方式来访问设备。由于嵌入式系统特殊的硬件配置, 设备驱动通常都是由应用开发者自己来编写, 而不是由设备生产商或操作系统厂商来开发。

在 Linux 操作系统下主要有两类设备: 字符设备和块设备。前者是以字节为单位逐个进行 I/O 操作的设备, 而后者则是以数据块来作为信息的存储和传输单位。作为操作系统内核与机器硬件之间的接口, 设备驱动程序的主要功能是:

- 对设备的初始化和释放。
- 把数据从内核传送到硬件和从硬件读取数据。
- 读取应用程序传送给设备文件的数据和回送应用程序请求的数据。这需要在用户空间、内核空间、总线以及外设之间传输数据。
- 检测和处理设备出现的错误。

从用户的角度出发, 如果在使用不同设备时, 需要使用不同的操作方法, 这样是非常麻烦的。因此, 用户希望能用同样的应用程序接口来访问设备和普通文件。Linux 抽象了对硬件的处理, 每一个硬件设备都使用一个特殊的设备文件来表示: 它们可以使用与文件相同的、标准的系统调用接口来完成打开、关闭、读、写和 I/O 控制等操作, 而驱动程序的主要任务就是要实现这些系统调用函数。这样一来, Linux 为文件和设备提供了一致的用户接口, 应用程序可以像操作普通文件一样对硬件设备进行操作。

每个设备文件都对应有两个设备号: 一个是主设备号, 标识该设备的种类, 也标识了该设备所使用的驱动程序; 另一个是次设备号, 标识使用同一设备驱动程序的不同硬件设备。

### (2) 设备驱动程序结构





Linux 的设备驱动程序与外界的接口主要有两部分：

- 向上的接口，即驱动程序与操作系统内核的接口。这是通过 `file_operations` 数据结构来完成的，这是一组通用的、标准的文件访问接口。
- 向下的接口，即驱动程序与设备的接口。这部分描述了驱动程序如何与设备进行交互，它与具体设备密切相关。

`file_operations` 数据结构中定义的接口函数主要包括以下几个。

- `open`：打开设备，准备进行 I/O 操作。
- `close`：关闭设备文件。
- `read`：从设备上读取数据。对于有缓冲区的 I/O 操作，一般是从缓冲区里读数据。
- `write`：往设备上写入数据，对于有缓冲区的 I/O 操作，一般是把数据写入缓冲区内。
- `lseek`：用来修改一个文件的当前读写位置，并将新位置作为正的返回值返回。
- `ioctl`：用来对设备进行控制。

当应用程序对一个设备文件进行上述这些操作时，Linux 内核将通过 `file_operations` 结构访问该设备的驱动程序所提供的相应函数。例如，当应用程序对设备文件执行读操作时，内核将调用相应的 `read` 函数去执行。

这样，在开发一个新设备的驱动程序时，主要的任务就是根据该设备的具体情况，从 `file_operations` 中的这些接口函数当中，选择一些对自己有意义的函数进行定义。例如，一个纯输入设备可能只有一个 `read()` 函数，而一个纯输出设备可能也只有一个 `write()` 函数。

### 3. 网络应用程序设计

在一些嵌入式操作系统中，支持 TCP/IP 网络协议软件栈，有些厂商甚至将 TCP/IP 协议栈用硬件方法予以实现。因此，程序员可以在此类嵌入式系统中，开发网络应用程序。

TCP/IP 协议簇包括 Socket 接口、传输层的 TCP 协议和 UDP 协议、网络层的 IP 协议、链路层接口及各种底层协议。Socket 是一个开发网络应用程序的编程接口，除了 TCP/IP 协议外，也适用于其他的一些协议。Socket 最初实现在 Berkeley UNIX 上，称为 BSD Socket。目前 BSD Socket 已经扩展到其他各种操作系统上，如 Windows 和 Linux 等，成为网络应用程序接口的事实标准。

编写一个基于 Socket 接口的网络应用程序的基本过程大致可以分为五个步骤。

#### (1) 初始化 Socket 网络应用程序的系统资源

在一个网络应用中要调用许多 Socket 函数，这些函数的功能是由嵌入式系统去具体实现的。因此，在网络应用程序中必须先对网络系统中的相关部分进行初始化，包括分配必要的资源，获取当前版本的一些细节情况等，而在应用程序退出之前，必须释放所有在初始化时分配的系统资源。一般来说，系统都会提供两个 API 函数：一个用于系统资源的初

始化，它必须在调用其他 Socket 函数之前被调用；另一个用于系统资源的释放，它必须在程序结束之前被调用。

### (2) 建立 Socket

完成初始化系统资源的工作后，便应当建立 Socket。Socket 一般译为“套接字”，可以把它理解为一个通信的端点。两个程序在通信之前，双方必须首先各自建立一个通信端点，否则是无法建立连接并相互通信的。

### (3) 建立连接

在建立一个 Socket 后，就得到了这个 Socket 的句柄。但是在利用这个 Socket 进行通信之前，对于面向连接的通信而言，还必须先建立一个从此 Socket 到目标机的连接。

### (4) 发送和接收数据

在面向连接的通信过程中，通信双方须建立并维护一个连接，所以在发送和接收数据时不用指明地址，只是将数据看成一个不间断的流，直接从已经建立的连接上发送或接收即可。

无连接的 Socket 在通信前无须建立连接，所以在通过一个无连接的 Socket 发送或接收数据时，必须每次都指明对方的主机地址，以便通知系统应把数据发往哪里或者要从哪里接收数据。

### (5) 关闭 Socket，释放系统资源

在应用程序使用完一个 Socket 后，应将这个 Socket 的句柄和占用的系统资源释放。

## 4.4.4 编码

### (1) 编码过程

在给定了软件设计规格说明书后，下一步的工作就是编写代码。一般来说，编码工作可以分为四个步骤：

① 确定源程序的标准格式，制定编程规范。

② 准备编程环境，包括软硬件平台的选择、操作系统、编程语言、集成开发环境等。

③ 编写代码。

④ 进行代码审查，以提高编码质量。为提高审查的效率，在代码审查前需要准备一份检查清单，并设定此次审查需找到的缺陷数量。在审查时，要检查软件规格说明书与编码内容是否一致；代码对硬件和操作系统资源的访问是否正确；中断控制模块是否正确，等等。

### (2) 编码准则

在嵌入式系统中，由于资源有限，且实时性和可靠性要求较高，因此，当我们在开发嵌入式软件时，要非常讲究，要注意对执行时间、存储空间和开发/维护时间这三种资源的

使用进行优化。也就是说，代码的执行速度要越快越好，系统占用的存储空间要越小越好，软件开发和维护的时间要越少越好。

具体来说，在编写代码时，需要做到以下几点。

- 保持函数短小精湛：一个函数应该只实现一个功能，如果函数的代码过于复杂，将多个功能混杂在一起，就很难具备可靠性和可维护性。另外，要限制函数的长度，一般来说，一个函数的长度最好不要超过 100 行。
- 封装代码：将数据以及对其进行操作的代码封装在一个实体中，其他代码不能直接访问这些数据。例如，全局变量必须在使用该变量的函数或模块内定义。对代码进行封装的结果就是消除了代码之间的依赖性，提高了对象的内聚性，使封装后的代码对其他行为的依赖性较小。
- 消除冗余代码：例如，将一个变量赋给它自己、初始化或设置一个变量后却从不使用它，等等。研究表明，即使是无害的冗余也往往与程序的缺陷高度关联。
- 减少实时代码：实时代码不但容易出错、编写成本较高，而且调试成本可能更高。如果可能，最好将对执行时间要求严格的代码转移到一个单独的任务或者程序段中。
- 编写优雅流畅的代码。
- 遵守代码编写标准并借助检查工具：用自动检验工具寻找缺陷比人工调试便宜，而且能捕捉到通过传统测试检查不到的各种问题。

### (3) 编码技术

#### ① 编程规范

在嵌入式软件开发过程中，遵守编程规范，养成良好的编程习惯，这是非常重要的，将直接影响到所编写的代码的质量。

编程规范主要涉及三方面的内容。

- 命名规则：从编译器的角度，一个合法的变量名由字母、数字和下划线三种字符组成，且第一个字符必须为字母或下划线。但是从程序员的角度，一个好的名字不仅要合法，还要载有足够的信息，做到“见名知意”，并且在语意清晰、不含歧义的前提下，尽可能简短。
- 编码格式：在程序书写时，要使用缩进（indentation）规则，例如变量的定义和可执行语句要缩进一个等级，当函数的参数过长时，也要缩进。另外，括号的使用要整齐配对，要善于使用空格和空行来美化代码。例如，在二元运算符与它的运算对象之间，要留有空格；在变量定义和代码之间要留有空行；在不同功能的代码段之间也要用空行隔开。
- 注释的书写：注释的典型内容包括函数的功能描述；设计过程中的决策，如数据

结构和算法的选择；错误的处理方式；复杂代码的设计思想等。在书写注释时要注意，注释的内容应该与相应的代码保持一致，同时要避免不必要的注释。

## ② 性能优化

由于嵌入式系统对实时性的要求较高，所以一般要求对代码的性能进行优化，使代码的执行速度越快越好。以算术运算为例，在编写代码时，需要仔细地选择和使用算术运算符。一般来说，整数的算术运算最快，其次是带有硬件支持的浮点运算，而用软件来实现的浮点运算是非常慢的。因此，在编码时要遵守以下准则：

- 尽量使用整数（char、short、int 和 long）的加法和减法。
- 如果没有硬件支持，尽量避免使用乘法。
- 尽量避免使用除法。
- 如果没有硬件支持，尽量避免使用浮点数。

图 4-28 所示是一个例子，有两段代码，它们的功能是完全一样的，都是对一个结构体数组的各个元素进行初始化，但采用两种不同的方法来实现。图 4-28（a）采用的是数组下标的方法，在定位第  $i$  个数组元素时，需要将  $i$  乘以结构体元素的大小，再加上数组的起始地址。图 4-28（b）采用的是指针访问的方法，先把指针  $fp$  初始化为数组的起始地址，然后每访问完一个数组元素，就把  $fp$  加 1，指向下一个元素。在一个采用 Pentium 4 作为 CPU 的 PC 机上，将这两段代码分别重复 10700 次，右边这段代码需要 1ms，而左边这段代码需要 2.13ms。

```
struct {  
    int a;  
    char b;  
    int c;  
} foo[10];  
int i;  
for (i = 0; i < 10; ++i)  
{  
    foo[i].a = 77;  
    foo[i].b = 88;  
    foo[i].c = 99;  
}
```

(a) 乘法

```
struct {  
    int a;  
    char b;  
    int c;  
} *fp, *fend, foo[10];  
fend = foo + 10;  
for (fp = foo; fp != fend; ++fp)  
{  
    fp->a = 77;  
    fp->b = 88;  
    fp->c = 99;  
}
```

(b) 加法

图 4-28 算术运算性能优化的例子

## 4.4.5 测试

### 1. 概述

软件测试是从经济和效率的角度出发,对软件代码进行质量和正确性保证的一个过程。软件测试是软件开发过程中的一个重要环节,也是从内部开发走向实际应用的关键一环。事实上,在计算机科学理论中有一个“停机定理”,它指出不可能证明任何一个程序是正确的,只要给予足够的测试,就能证明一个程序是错误的。因此,在软件开发中,要用精心设计和量化的测试计划来进行测试,以降低程序员自身、公司及客户的风险。对于关键性的系统和关键性的软件模块,尤其要重视测试工作。例如,医院中的放射性治疗仪,如果出现问题,可能会造成巨大的医疗事故。从工程实践的经验来看,软件中的错误发现得越早,相应的修改费用就越低。反之,如果在已经发布的软件产品中发现缺陷及错误,那么修改的成本和代价就非常高。另外,通过测试,能够找到并清除死代码及无效代码,并确认硬件的潜力是否已得到了充分发挥,从而进一步提高系统的性能。

嵌入式软件的测试工作与台式机上的应用软件的测试工作有许多共同之处,但也有很大的区别,这些区别是由嵌入式软件的特点所造成的。

- 嵌入式软件必须在很长一段时间内稳定的运行。
- 嵌入式软件一般不会频繁地由用户进行升级。
- 嵌入式软件可能使用在一些关键性的应用产品中,如医院中的放射性治疗仪、工厂中的自动化控制设备、核电站的控制系统、军事中的武器控制系统等。
- 嵌入式软件必须与嵌入式硬件一起对产品的故障负责。
- 真实世界中的事件一般是异步而且不可预测的,这就使模拟测试既困难又不可靠。

正是由于这些特点,使得嵌入式软件的测试工作与普通的应用软件的测试有如下的一些区别。

- 嵌入式系统的硬件一般采用专门的测试仪器进行测试,而这些测试工具一般不会在应用软件的开发中使用。
- 由于嵌入式软件自身的特点,其测试过程相当复杂,传统的软件测试理论不能直接用于嵌入式软件的测试,如果没有一个很好的除错环境,那就只能依靠人力和程序员的经验来进行。在嵌入式系统中,由于在目标平台上通常没有很好的显示或输出能力,所以经常采用交叉编译和交叉测试的方法,常常需要在基于目标机的测试和基于宿主机的测试之间做出折中。基于目标机的测试需要消耗较多的时间和费用,而基于宿主机的测试虽然代价较小,但毕竟是在仿真环境中进行,难以完全反映软件运行时的真实情况。这两种环境下的测试可以发现不同的软件缺陷,关键是要对目标机环境和宿主机环境下的测试内容进行合理取舍。而对于普

通的应用软件，它的测试环境可能就是它的开发环境，同时也是它的运行环境。

- 与 PC 软件相比，在测试嵌入式软件时，除了逻辑上的正确性之外，还要看重系统的性能和健壮性。原因有两点：首先，由于嵌入式环境的资源有限，CPU 主频低，内存小，只有少量的存储空间等；其次，嵌入式软件一般都固化在存储介质上，不易修改升级，而且运行环境比较恶劣，可能会发生断电、物理损坏等异常情形。
- 嵌入式软件的一个重要特点就是实时性，也就是说，嵌入式软件的执行要满足一定的时间约束。在嵌入式系统中，应用软件自身算法的复杂度和操作系统的任务调度，决定了系统资源的分配和消耗。因此，对系统的实时性进行测试时，要借助一定的测试工具对应用程序的算法复杂度和操作系统的任务调度进行分析测试。
- 嵌入式系统的开发是一个软硬件互相协调、互相反馈和互相测试的过程。由于嵌入式软件对硬件的依赖性要求，在进行软件测试时必须最大限度地模拟被测软件的实际运行环境，以保证测试的可靠性。但这种模拟有时是不太可靠的，这就增加了测试的难度，使得在系统测试时，错误的定位较为困难。

## 2. 测试的任务

系统测试的目的是为了发现至今尚未发现的错误。图 4-29 所示是一个嵌入式软件的统一测试模型。

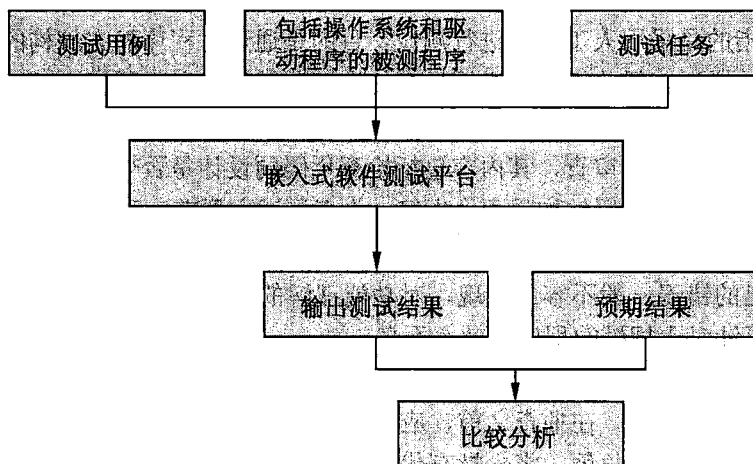


图 4-29 嵌入式软件的统一测试模型

一个规范化的测试过程通常包括以下的基本测试活动。

- (1) 拟定测试计划。在制定测试计划时，要充分考虑整个项目的开发时间和开发进度



以及一些人为因素和客观条件等,使得测试计划是可行的。测试计划的内容主要有测试的内容、进度安排、测试所需要的环境和条件、测试培训安排等。

(2) 编制测试大纲。测试大纲是测试的依据,它明确、详尽地规定了在测试中针对系统的每一项功能或特性所必须完成的基本测试项目和测试完成的标准。

(3) 根据测试大纲设计和生成测试用例。在设计测试用例的时候,不仅要设计有效合理的输入条件,也要包含不合理、失效的输入条件。在测试时,人们往往习惯按照合理的、正常的情况进行测试,而忽略了对异常、不合理、意想不到的情况进行测试,而这些可能就是隐患。在设计好测试用例后,要产生相应的测试设计说明文档,其内容主要有被测项目、输入数据、测试过程、预期输出结果等。

(4) 实施测试。测试的实施阶段是由一系列的测试周期组成的。在每个测试周期中,测试人员和开发人员将依据预先编制好的测试大纲和准备好的测试用例,对被测软件或设备进行完整的测试。

(5) 生成测试报告。测试完成后,要形成相应的测试报告。这主要用于对测试进行概要说明,列出测试的结论,指出缺陷和错误。另外,给出一些建议,如可采用的修改方法、各项修改预计的工作量及修改的负责人员等。

### 3. 测试的方法

软件测试方法可以分为两大类:人工测试和机器测试。

#### (1) 人工测试

人工测试指的是采用人工方式进行测试,目的是通过对程序静态结构的检查,找出编译时不能发现的错误。经验表明,组织良好的人工测试可以发现程序中 30%~70% 的编码和逻辑设计错误。

人工测试又称为代码审查,其内容包括检查代码同设计是否一致、检查代码逻辑表达是否正确和完整、检查代码结构是否合理等。人工测试主要有三种方法。

- 个人复查:指程序员本人对程序进行检查。由于心理上的原因和思维惯性的影响,对自己的错误一般不容易发现,对功能理解的错误更不可能纠正。因此,这种方法主要针对小规模的程序,效率不高。
- 抽查:通常 3~5 人组成测试小组,测试人员应该是有参加过该项目开发的有经验的程序设计人员。在抽查之前,应先阅读相关的软件资料和源程序,然后测试人员扮演计算机的角色,将一批有代表性的测试数据沿着程序的逻辑走一遍,监视程序的执行情况。人工检测程序很慢,只能选择少量简单的例子。
- 会审:测试人员的构成与抽查类似。在会审之前,测试人员应该充分阅读相关资料,如系统分析说明书、系统设计说明书、源程序等。有经验的测试人员会列出尽可能多的典型错误,在会审时,由编程人员逐句讲解程序,测试人员逐个审查、

提问。通过这种方式,往往可能使编程人员发现自己以前没有意识到的错误,使问题暴露。会审后,要将发现的问题登记、分析、归类。

代码复审应该在被测软件编译成功之后,编译都没通过的软件当然谈不上复审。在复审期间,应保证有足够的时间,让测试小组对问题进行充分讨论,这样才能有效提高测试的效率,避免出错。

## (2) 机器测试

机器测试是把设计好的测试用例作用于被测程序,比较测试结果和预期结果是否一致,如果不一致,就说明可能存在错误。机器测试只能发现错误的症状,但无法对问题进行定位。

机器测试分为黑盒测试和白盒测试两种。近年来在软件测试中,测试代码的覆盖率逐渐成为软件测试的评价标准。不管采用何种测试方法,都要尽可能地提高软件测试中的代码覆盖率。在理想情况下,应该要测试程序中每一种可能的行为,即对于所有的输入组合和所有可能的判定路径,都要至少测试一次。但这是不切实际的,例如,在 Glen Ford Myers 的 *The Art of Software Testing* 一书中,他描述了一个仅有五个判定点的小程序,但它却有  $10^{14}$  条独一无二的执行路径。如果编写、执行及编译一个测试用例要用五分钟,那么彻底测试这个程序将要用 10 亿年。因此,在一般情形下,理想状态是不能达到的,只能使用近似理想状态的方法,揭露出尽可能多的错误。

### ① 黑盒测试

黑盒测试也称为功能测试。将软件看成黑盒子,在完全不考虑软件的内部结构和特性的情况下,测试软件的外部特性。嵌入式系统包含输入、输出以及在两者之间实现的一些算法。黑盒测试关心的是在系统中哪些输入是可以接受的,这些输入将产生什么样的输出,它并不关心输入与输出之间的算法到底是如何实现的。一般来说,黑盒测试包括以下几点。

- 极限情况测试:在测试中有意使输入信道、内存缓冲区、磁盘控制器和内存管理器等部件超载。
- 边界测试:对输入值进行设计,使得输入或输出值落在边界范围内。例如,对于整型输入,应选择最大值、最小值、-1、0 和 1 等边界值;再如,输入某些特定的值,使得输出值位于最小值或最大值等边界上。
- 异常测试:有意识地去触发系统的失败模式或异常模式。
- 错误猜测:根据以前进行软件测试及测试类似程序的工作经验,选择一些容易引发错误的测试用例。
- 随机测试:随机地进行测试。这种方法的效率比较低,主要用在用户界面的测试上,用来评估界面代码的健壮性。
- 性能测试:由于性能要求是产品需求的一部分,所以性能测试也处于功能测试的



范围之内。

由于黑盒测试仅仅依赖于程序的需求及其功能行为，所以在系统的需求工作完成后就可以马上进行测试用例的设计，使测试用例的开发与系统其余部分的设计同时进行。

## ② 白盒测试

白盒测试也称为结构测试。它将软件看成透明的白盒，根据程序的内部结构和逻辑来设计测试用例，对程序的路径和过程进行测试，检查是否满足设计的需要。其原则是：

- 程序模块中的所有独立路径至少执行一次。
- 在所有的逻辑判断中，取“真”和取“假”的两种情况至少都能执行一次。
- 每个循环都应在边界条件和一般条件下各执行一次。
- 测试程序内部的数据结构的有效性等。

从嵌入式系统的观点来看，白盒测试是很重要的一种测试方法，测试人员可以很清楚已经有多少代码被检查过了。而且通过测试结果，可以准确地预测出系统中的程序设计错误的数量规模。另外，从理论上来说，白盒测试能够利用它所需要的一切信息来进行测试。例如，白盒测试可能使用 JTAG 端口，把特定内存地址的值作为测试的一部分。

## 4. 测试的步骤

嵌入式软件的测试可以分为五个步骤。

### (1) 系统平台测试

包括硬件电路测试、操作系统及底层驱动程序的测试等。硬件电路需要用专门的测试工具来进行测试；操作系统和底层驱动程序的测试主要包括测试操作系统的任务调度、实时性能、通信端口的数据传输率等。该阶段测试完成后，系统应为一个完整的嵌入式系统平台。

### (2) 单元测试

单元测试也称为模块测试。一般来说，一个大型的嵌入式软件系统会被划分为若干个较小的任务模块，由不同的程序员负责，同时进行编码。在各个模块编写完成且无编译错误后，在把它们集成起来之前，必须对各个模块分别进行测试。这个阶段的测试一般是在宿主机上进行的，因为宿主机上有丰富的资源和方便的调试环境。单元测试一般采用的是白盒测试法，要尽可能地测试每一个函数、每一个条件分支、每一个程序语句，以提高代码测试的覆盖率。

单元测试主要从模块的以下五个特征着手进行检查。

- 模块接口。模块的接口保证了测试模块的数据流可以正确地流入、流出。在测试中应检查以下要点：在模块调用或函数调用中，参数的使用是否正确；全局变量在各模块中的定义和用法是否一致；开/关语句、I/O 语句和文件的使用是否正确等。

- 局部数据结构。在单元测试中，局部数据结构出错是比较常见的错误，在测试时应重点考虑以下因素：变量的说明是否合适；是否使用了尚未赋值或尚未初始化的变量；变量的初始值是否正确；变量名是否有错（如拼写错误）等。
- 重要的执行路径。在单元测试中，对路径的测试是最基本的任务。由于不能进行穷举测试，需要精心设计测试用例来发现是否有计算、比较或控制流等方面的错误。例如，算术运算的优先次序不正确或理解错误；精度不够；运算对象的类型不匹配；逻辑运算符不正确或优先次序错误；循环终止条件不正确；分支循环的出口错误等。
- 出错处理。好的设计应该能预测到出错的条件并且有出错处理的途径。虽然计算机可以显示出错信息的内容，但仍需要程序员对出错进行处理，保证其逻辑的正确性，以便于用户维护。
- 边界条件。边界条件的测试是单元测试的最后工作，也是容易出错的地方。

由于模块不是独立运行的程序，且各模块之间存在调用与被调用的关系。因此，在对各个模块进行测试时，需要开发两种模块。

- 驱动模块：相当于一个主程序，负责接收测试用例的数据，将这些数据送到测试模块，并输出测试结果。
- 桩模块：也称为存根模块。用来代替测试模块中所调用的子模块，其内可进行少量的数据处理，目的是为了检验入口，并输出调用和返回的信息。

提高模块的内聚度可以简化单元测试。如果每个模块只完成一种功能，对于具体模块来说，所需的测试方案数据就会显著减少，而且更容易发现和预测模块中的错误。

### （3）集成测试

集成测试也称为组装测试，就是把各个模块按照系统设计说明书的要求组合起来进行测试。即使所有模块都通过了各自的单元测试，但在组装之后，仍可能会出现下列问题：

- 穿过模块的数据丢失。
- 一个模块的功能对其他模块造成有害的影响。
- 各个模块组装起来没有达到预期的功能。
- 全局数据结构出现问题。
- 单个模块的误差可以接受，但模块组合后，可能会出现误差的累积，最后到达不能接受的程度。

集成测试通常有两种方法：一种是分别测试各个模块，再把这些模块组合起来进行整体测试，即非增量式集成；另一种是把下一个要测试的模块组合到已测试好的模块中，测试完后再将下一个模块组合进来，进行测试，这样逐步地把所有的模块组合在一起，并完成测试，即增量式集成。非增量集成可以对模块进行并行测试，能充分利用人力，加快测

试进度,但这种方法容易造成混乱,出现错误时不容易查找和定位;而增量式测试的范围一步步扩大,错误容易定位,而且已测试的模块可在新的条件下再测试,测试更彻底。

嵌入式软件的集成测试可以在宿主机上进行,采用黑盒与白盒相结合的方法进行测试,要最大限度地模拟实际运行环境。为了提高测试效率,可以暂时屏蔽一些不影响系统执行的函数,以及一些数据传递难以模拟的函数。

#### (4) 系统测试

系统测试是将嵌入式软件、硬件、外设和网络等各种因素结合在一起,进行整个系统的组装测试和确认测试。这个阶段的测试一般是在目标机上进行的,需要把系统移植到目标机上来。在现场环境中,从用户的角度对系统进行黑盒测试,验证每一项具体的功能,并与系统的需求进行比较,发现所开发的系统与用户需求不符或矛盾的地方。系统测试要根据系统方案说明书来设计测试用例,常见的系统测试主要有以下内容。

- 恢复测试:恢复测试监测系统的容错能力。它的检测方法是采用各种方法让系统出现故障,然后检验系统是否能按照要求从故障中恢复过来,并在约定的时间内开始事务处理,且不对系统造成任何伤害。如果系统的恢复是自动的,需要验证重新初始化、检查点、数据恢复等是否正确;如果恢复需要人工干预,就要对恢复的平均时间进行评估并判断它是否在允许的范围内。
- 强度测试:是对系统在异常情况下的承受能力的测试,是检查系统在极限状态下运行时,性能下降的幅度是否在允许的范围内。因此,强度测试要求系统在非正常数量、频率或容量的情况下运行。强度测试主要是为了发现在有效的输入数据中可能引起不稳定或不正确的数据组合,例如,让系统传输超过最大设计能力的的数据,包括内存的写入和读出等。
- 性能测试:检查系统是否满足设计方案说明书中对性能的要求。性能测试覆盖了软件测试的各个阶段,而不是等到系统的各部分都组装之后,才确定系统的真正性能。它通常与强度测试结合起来进行,并同时软件、硬件进行测试。软件方面主要从响应时间、处理速度、吞吐量、处理精度等方面来检测。
- 可靠性测试:通常使用两个指标来衡量系统的可靠性——平均失效间隔时间是否超过了规定的时限;因故障而停机的时间在一年中不应超过多少时间。

#### (5) 测试结果分析

对测试结果进行分析和比较可以帮助错误的定位,指导程序员修改代码,并指明进一步测试的方向。测试结果分析是一次测试的最后环节,在分析时应该考虑软件的运行环境和实际运行环境的差异以及各种外界因素的影响等。

### 5. 覆盖测试

#### (1) 覆盖测试概述

覆盖测试是一种白盒测试方法,测试人员必须拥有程序的规格说明书和程序清单。它的基本思路是以程序的内部结构为基础来设计测试用例,以覆盖尽可能多的程序内部逻辑结构,发现其中的错误和问题。覆盖测试一般用在软件测试的早期,即单元测试阶段。

覆盖测试的主要策略包括以下五项。

- 语句覆盖:在设计测试用例时,使程序中的每一条语句都至少执行一次。
- 判定覆盖:执行足够多的测试用例,使程序中的每个判定都获得一次“真”值和“假”值,即每一个分支都至少执行1次。
- 条件覆盖:执行足够多的测试用例,使判定中的每个条件都获得所有可能的逻辑值。
- 判定/条件覆盖:执行足够多的测试用例,使每个分支都至少执行一次,且判定中的每个条件都获得所有可能的逻辑值。
- 条件组合覆盖:执行足够多的测试用例,使每个判定中的各种条件组合都至少出现一次。其特点是覆盖较充分,满足条件组合覆盖的测试用例也一定满足判定覆盖、条件覆盖和判定/条件覆盖。

其他一些覆盖策略还包括:修改的条件/判定覆盖、路径覆盖、函数覆盖、调用覆盖、线性代码顺序和跳转覆盖、数据流覆盖、目标代码分支覆盖、循环覆盖、关系操作符覆盖等。随着软件规模的增长,实现全面覆盖所需的测试用例的数目也越来越庞大,所以根据被测软件对象的特点来选择适当的覆盖策略是非常重要的。同时,要确定合理的测试目标,并与形式化评审等方法相结合,以发现更多的软件故障。

## (2) 覆盖测试工具

要取得较好的覆盖测试效果,需要借助一定的工具软件。这些工具软件一般具备如下功能特点,可弥补人为测试的缺陷。

- 分析软件内部结构,帮助制定覆盖策略及设计测试用例。
- 与适当的编译器相结合,对被测软件实施自动插装,以便在其运行过程中生成覆盖信息并收集这些信息。
- 根据收集的覆盖信息计算覆盖率,帮助测试人员找到未被覆盖的软件部位,以改进测试用例,提高覆盖率。

在利用工具进行动态覆盖测试时,需要三个要素:测试用例、插装过的被测代码、搜集覆盖信息并进行分析的工具本身。代码插装由工具自动完成,通过执行测试用例,再由工具收集覆盖信息并进行分析,即可看到覆盖率指标。图4-30所示展示了实现覆盖测试的基本过程。

## (3) 嵌入式软件的覆盖测试原理

与通用软件不同,嵌入式软件采用的是交叉开发的方式:开发工具运行在软硬件配置

丰富的宿主机上，而嵌入式应用程序则运行在软硬件资源相对缺乏的目标机上。对于这类软件，在测试时也存在同样的问题：测试工具运行在宿主机上，测试所需要的信息却在目标机上产生，并通过一定的物理/逻辑连接传输到宿主机上，被测试工具接收。因此，嵌入式软件测试的一个重要问题是建立宿主机与目标机之间的物理/逻辑连接，解决数据信息的传输问题。

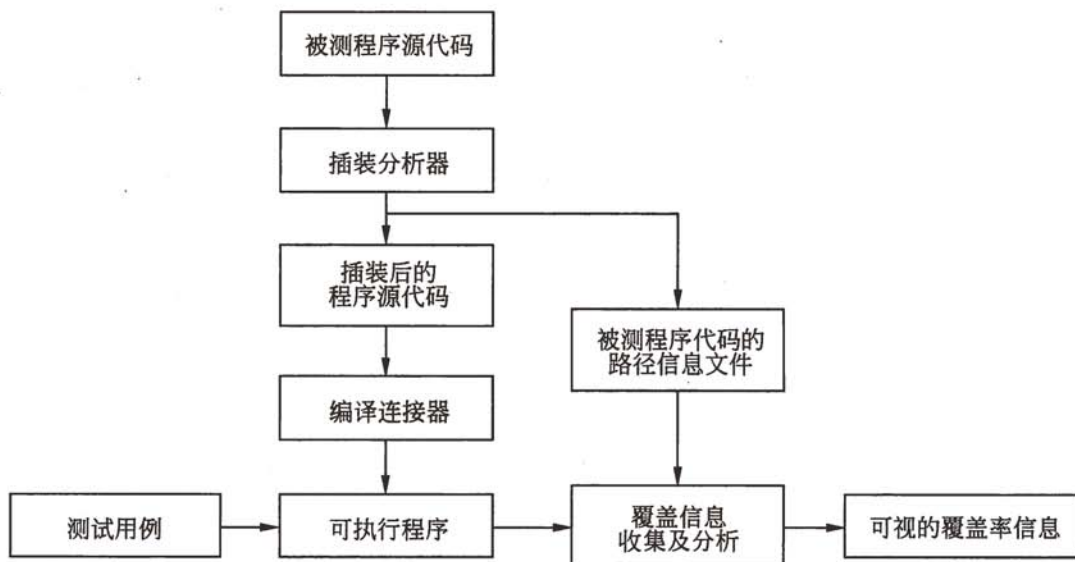


图 4-30 覆盖测试的基本过程

嵌入式软件覆盖测试的基本原理如图 4-31 所示。在目标机端，插装过的被测应用程序将覆盖信息发送到消息队列中，一个专门的任务负责在适当的时候将这些信息发送到宿主机端。宿主机端有专门的模块负责接收覆盖信息，并交给分析工具分析和在线动态显示覆盖率的增长情况。

#### 4.4.6 下载和运行

如前所述，嵌入式系统采用宿主机/目标机模式来开发嵌入式应用软件，然后通过串口或网络等通信线路，将交叉编译生成的目标代码传输并装载到目标机上，在监控程序或操作系统的支持下利用交叉调试器进行分析和调试，最后目标机可以在特定环境下脱离宿主机单独运行，如图 4-32 所示。

根据嵌入式系统硬件的配置情况，固化的方式有多种，可以固化在 EEPROM 和 Flash 这类存储器中，也可以固化在 DOC 和 DOM 等电子盘中。比较常见的方式，还是使用编程

器将二进制映像文件写入到目标机的 EEPROM 或 Flash 中, 或者是使用 TFTP 协议进行远程文件传送。

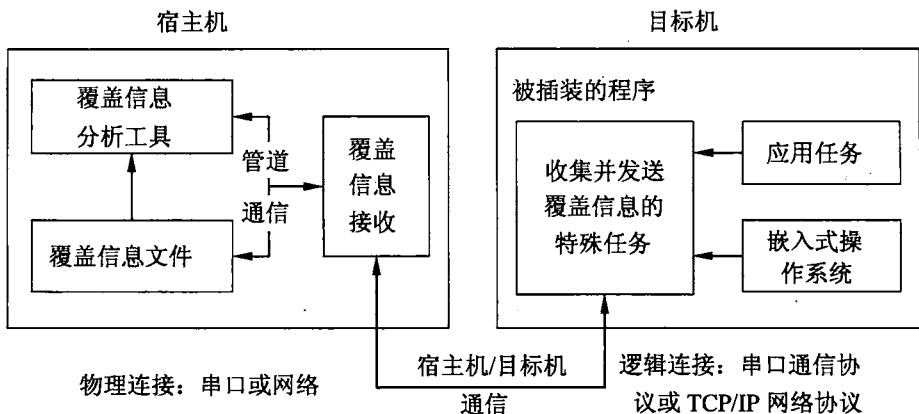


图 4-31 嵌入式软件覆盖测试的基本原理

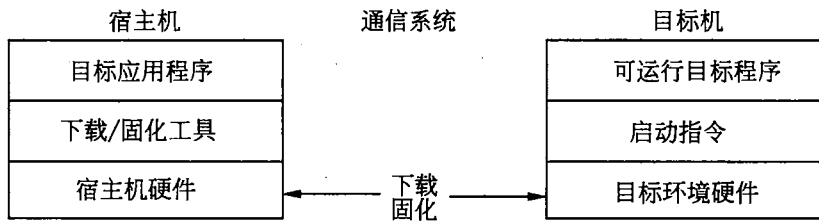


图 4-32 嵌入式应用程序下载/固化

编程器上面有各种形状和大小不同的芯片插座, 可以通过通信线路将其与宿主机连在一起。在进行固化时, 一般是先把存储芯片插入编程器上某个大小、形状合适的插座上, 并通过软件选择芯片的型号, 然后将被固化的程序文件传到编程器上。整个固化过程可能需要几秒钟到几分钟, 这要看文件的大小和所用的芯片型号。

简单文件传输协议 (Trivial File Transfer Protocol, TFTP) 可以看成是一个简化了的 FTP, 主要用于下载映像文件。它同 FTP 的主要区别是没有用户权限管理的功能, 也就是说, TFTP 不需要认证客户端的权限。这样, 远程启动的目标板在启动一个完整的操作系统之前, 就可以通过 TFTP 下载启动映像文件, 而不需要证明自己是合法的用户。一般来说, 在目标机初次配置时, 需要通过 BootLoader 的 TFTP 客户端下载启动映像——这个映像包含了嵌入式操作系统和应用程序代码, 然后将下载得到的映像烧写至闪存, 这样每次启动时就可以直接从 Flash 中载入了。当然, 在有些嵌入式系统中, 把嵌入式操作系统也

固化在目标机的 ROM 中,然后把各个应用程序做成可加载的模块。当目标机操作系统启动后,可以根据自己的需要,从宿主机下载相应的应用程序模块。

## 4.5 嵌入式软件移植

嵌入式软件与通用软件的不同在于,嵌入式软件高度依赖于目标应用的软硬件环境。软件的部分功能函数由汇编语言完成,与处理器密切相关,可移植性差。一般来说,嵌入式应用软件追求正确性、实时性,所以使用编译效率高的汇编语言有时是难以避免的,但这就会导致应用软件的可移植性大打折扣。另外,对于一个运行良好的嵌入式软件或其中的部分子程序,在今后的开发中可能会再次用在类似的应用领域。由于原有的代码已被反复应用、测试和维护,具有很好的稳定性。因此,在原有代码的基础上进行移植将会缩短开发周期,提高开发效率,降低开发成本。基于以上原因,在嵌入式软件开发中,必须高度关注应用软件的可移植性和可重用性。

不过,可移植性和可重用性的程度应该根据实际的应用情况来考虑。由于嵌入式应用软件有许多自身的特点,如果追求过高的可移植性和可重用性,可能会恶化应用软件的实时性能,并增加软件的代码量。这对于资源本来就有限的嵌入式应用环境来说,是得不偿失的。尽管如此,我们仍然可以在资源有限、满足系统需求的情况下,尽可能地把可移植性和可重用性作为第二目标,致力于开发正确性、实时性、代码量、可移植性和可重用性相对均衡的嵌入式应用软件。

嵌入式应用软件的开发可以分为两种情况:无操作系统的情况和有操作系统的情况。与之相对应,在移植嵌入式软件的时候,也可以分两种情况来讨论,即无操作系统的软件移植和有操作系统的软件移植。对于后者,又可以细分为两种情况:把操作系统和应用软件作为一个整体进行移植;把应用软件移植到一个新的操作系统上。

### 4.5.1 无操作系统的软件移植

如果在一个嵌入式系统中,软件的开发是直接硬件平台的基础上进行的,没有使用操作系统,那么当处理器等硬件设备发生变化时,需要把原有的应用软件移植到新的硬件平台上运行。一般来说,对于这一类的嵌入式系统,它们的应用软件通常都比较简单,软件的代码量不是很大。在移植的时候,如果编程语言使用的是与处理器密切相关的汇编语言,那么移植将会变得非常困难,甚至是不可能的。此时的移植类似于重新开发,当然在数据结构和算法的层面上还是可以重用的,但我们不考虑这种情况。

如果软件大部分是用 C 语言开发的,那么可以考虑移植问题。一个好的程序设计应该是模块化和层次化的,而 C 语言的最大优点是可移植性比较好。基于层次化的嵌入式应用

软件通常设计成如图 4-33 所示的结构。

在图 4-33 所示结构中，软件可分为两层结构，I/O 模块属于设备驱动程序层，它以嵌入式硬件为运行平台，实现了各种 I/O 设备的 I/O 功能，并向上层的应用软件提供了相应的 I/O 接口函数 API，如控制功能、数据读写等。这些接口函数被设计成与硬件无关的，所以在移植系统时，只需要重新编写与处理器有关的 I/O 模块即可，不需要修改该模块的 API。这时，移植的工作量主要体现在 I/O 的编码工作上。

可以对上述软件结构做进一步的细化设计，如图 4-34 所示。在这种体系结构中，在处理器的硬件层之上添加了一个硬件抽象层，这层软件把不同类型的硬件进行了封装和抽象。对于 I/O 模块，它不再是直接面对处理器硬件，而是基于硬件抽象层。也就是说，它被设计为与硬件无关的。这样的三层软件结构的优点是需要移植的代码进一步减少，移植的工作量也进一步减少。

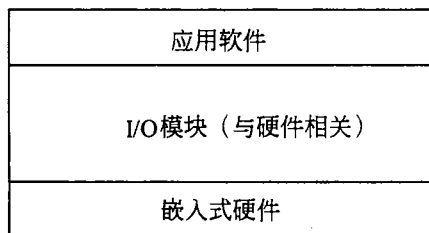


图 4-33 基于模块化的嵌入式软件结构

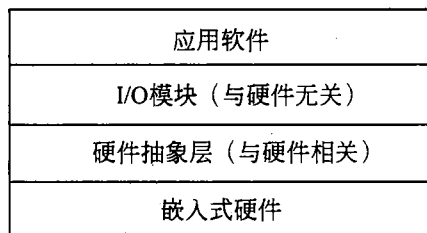


图 4-34 具有硬件抽象层的软件结构

## 4.5.2 有操作系统的软件移植

有操作系统的软件移植讨论的是把操作系统和应用软件作为一个整体，移植到一个新的嵌入式硬件平台上。

在 3.1.3 节中，曾经将嵌入式软件的体系结构分为四个层次，即设备驱动层、操作系统层、中间件层和应用软件层。当我们要把一个嵌入式软件系统整体移植到一个新的硬件平台时，真正需要移植的是与硬件直接打交道的部分，包括设备驱动层的软件和操作系统当中的部分代码，而其他的软件，如操作系统内核、中间件和应用软件，不用做任何修改。当然，在有些嵌入式操作系统中，如单体结构的操作系统，把设备驱动层的软件也集成在系统内核中，这时就要把相应的软件摘取出来进行修改。总之，在系统移植时，真正需要移植的主要是：引导加载程序 BootLoader、设备驱动程序以及操作系统中同处理器密切相关的代码。

为提高可移植性，BootLoader 的实现一般分为 stage1 和 stage2 两大部分。依赖于 CPU 体系结构的代码，如设备初始化代码等，通常都放在 stage1 中，用汇编语言来实现；而



stage2 则采用 C 语言来实现。在移植时, 主要的工作量在 stage1 的移植上, 基本上要重新编写。

一般来说, 一个嵌入式操作系统在设计的时候, 就已经充分考虑了可移植性, 所以它的移植相对来说是比较容易的。以  $\mu\text{C}/\text{OS-II}$  为例, 为了方便移植,  $\mu\text{C}/\text{OS-II}$  的大部分代码都是用标准的 C 语言编写的, 不需要改动, 只有少部分代码, 尤其是那些跟 CPU 寄存器打交道的代码, 需要针对具体的 CPU 类型进行修改, 这些代码一般都是用汇编语言来写的。

如图 4-35 所示,  $\mu\text{C}/\text{OS-II}$  操作系统的代码被分为三大部分: 第一部分是与处理器无关的代码, 如任务管理、任务调度、存储管理、信号量、邮箱、消息队列等; 第二部分与系统的配置有关, 应用程序开发人员可以通过修改这些配置文件来裁剪内核, 选择自己需要的系统服务; 第三部分是与处理器相关的代码, 包括 `OS_CPU.H`、`OS_CPU_A.ASM` 和 `OS_CPU_C.C` 三个文件, 在移植  $\mu\text{C}/\text{OS-II}$  操作系统时, 主要修改的就是这三个文件。

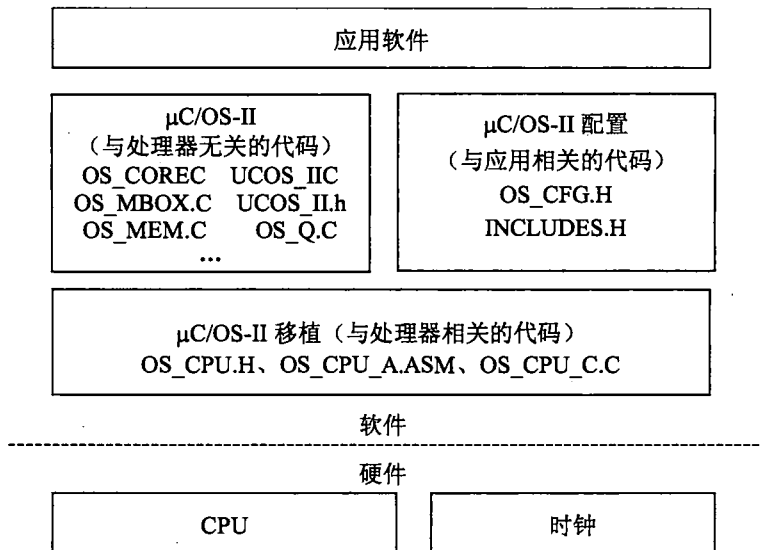


图 4-35  $\mu\text{C}/\text{OS-II}$  移植的示意图

- `OS_CPU.H`: 包括三部分的内容——一个符号常量, 用来设定处理器的栈的增长方向; 三个宏定义, 用来关闭和打开中断; 10 个数据类型的定义, 用来定义与编译器无关的数据类型 (在操作系统内部, 只使用这些数据类型, 而不使用标准 C 的数据类型)。
- `OS_CPU.ASM`: 用汇编语言编写 `OS_CPU.ASM` 文件中的四个与处理器相关的函数, 包括任务切换、时钟中断服务程序等。

- OS\_CPU\_C.C: 用 C 语言编写的 OS\_CPU\_C.C 文件中的 10 个与操作系统相关的函数, 一般只需要改写其中的一个函数, 即任务堆栈的初始化函数。

根据目标处理器的不同, 一个移植实例可能需要编写或改写 50 行~300 行的代码, 需要的时间从几个小时到一个星期不等。

### 4.5.3 应用软件的移植

嵌入式应用软件的移植指的是把应用软件从一个嵌入式操作系统平台移植到另一个操作系统平台。

一个应用软件的实现涉及两个方面的问题:

- 这个应用软件必须用某种编程语言来编写, 如汇编语言、C 语言、C++ 语言。
- 这个应用软件必须在某一个平台上运行, 这个平台一般是一个操作系统, 如 Windows XP、Linux 等。

当然, 也有一些软件系统, 它们既是编程语言, 又是运行平台, 如 Java。

因此, 当我们在移植一个应用软件的时候, 既要考虑编程语言的因素, 也要考虑运行平台的因素。对于 PC 机上的应用软件来说, 它的运行平台比较有限, 只有两大类, 即 Windows 系列和 UNIX 系列, 前者以 Windows XP 为代表, 后者以 Linux 为代表。相应的, 每一类平台都有各自的一套应用程序编程接口 (Application Programming Interface, API)。在嵌入式系统中, 编程语言的问题不大, 因为大多数嵌入式开发都是采用移植性较好的 C 语言。但是在运行平台上, 嵌入式操作系统的选择是非常多的, 目前已经开发了数以百计的各具特色的嵌入式操作系统产品。从理论上说, 每一个操作系统都会定义一组 API 接口函数, 因此, 如果要在嵌入式平台上进行应用软件的移植, 难度是比较大的。

为了提高嵌入式应用软件的移植性, 在软件开发时需要遵守以下的一些原则:

- 在软件设计上, 要采用层次化设计和模块化设计。所谓层次化, 指的是软件设计的纵向结构, 下层为上层提供服务, 上层去调用下层提供的服务。每一个层次都应该定义清晰的接口和功能, 分层的数量要合适。层次化结构设计的优点是: 在进行系统移植时, 通常只需要修改底层软件, 不需要去修改上层软件。所谓模块化, 既体现在整体软件的设计上, 又体现在同一层的软件结构上。模块化不同于层次化, 一般来说, 软件模块之间是相互独立的, 一个模块的实现不依赖于其他模块的实现。良好的模块化设计, 可以保证很容易地进行软件模块的裁减和更新。
- 在软件体系结构上, 可以在操作系统和应用软件之间引入一个虚拟机层, 或者叫操作系统抽象层, 把一些通用的、共性的操作系统 API 接口函数封装起来。当我们在编写一个应用程序时, 不是直接去调用实际操作系统的 API, 而是使用虚拟层所提供的 API。这样, 在移植这个应用程序的时候, 只要针对新的操作系统平台,



去实现这个虚拟层即可，其他的代码不用做任何的修改。在定义这个虚拟层时，要综合考虑现有的各种嵌入式操作系统的功能和特性，尽量采用标准的操作系统接口，如 POSIX 标准。

- 在功能服务的调用上，要尽量使用可移植的函数，如标准的 C 语言函数，或自己编写的函数；尽量不要使用依赖于特定操作系统的 API 函数。
- 在数据类型上，由于 C 语言的数据类型与机器的字长和编译器有关，所以可以用宏定义的方式来定义一组可移植的数据类型，然后在应用程序的内部，只使用这些数据类型，而不使用 C 语言的数据类型。例如，可以用 INT32U 来表示无符号的 32 位整型数据，对于实际的编译器，可以定义为 `#define INT32U unsigned int`。
- 将不可移植的部分局域化。对于想进行软件移植的程序设计人员来说，如果应用程序的各个地方都散布着不可移植的代码，就必须从软件中一一找出它们，然后修改。这将是一件非常费时又费力的事情，而且这种修改也容易导致新的问题。为了提高移植的效率，可以把不可移植的代码通过宏定义和函数的形式，分类集中于某几个特定的文件之中。这样，对不可移植代码的使用，就可转换成对函数和宏定义的使用，在以后的移植过程中，既有利于迅速地需要对需要修改的代码进行定位，又可方便地进行修改。
- 提高代码的可重用性。在进行嵌入式软件开发时，要有意识地提高代码的可重用性，不断积累可重用的软件资源。例如，可以更好地抽象软件的函数，使之更加模块化，功能更专一，接口更简洁明了。

## 第5章 嵌入式系统开发与维护知识

### 5.1 系统开发过程及其项目管理

嵌入式系统的设计和开发必须将所有的硬件、软件、人力资源等集中起来，并且进行适当的组合以实现目标系统对性能和功能等各方面的需求。在嵌入式系统的开发过程当中，实时性、可靠性、功耗等都与功能一样重要，这就是使嵌入式系统的开发关注的方面更广泛，要求的精度也更高。

嵌入式系统的设计和开发流程与通用系统的开发流程大同小异，但开发所使用的设计方法有一定的差异。根据嵌入式系统设计的特点和系统的组成，嵌入式系统设计一般分为以下几个阶段：需求分析阶段、规格说明阶段、体系结构设计阶段、设计硬件构件和软件构件阶段、系统集成阶段。各个阶段通常需要不断的反复和修改，直到完成最终设计目标。

#### 5.1.1 系统开发生命周期各阶段的目标和任务的划分方法

##### 1. 常用开发模型

软件开发模型（Software Development Model, SDM）是指软件开发全部过程、活动和任务的结构框架。软件开发包括需求、设计、编码和测试等阶段，有时也包括维护阶段。

典型的开发模型有：边做边修改模型、瀑布模型、渐增模型/演化/迭代、原型模型、螺旋模型。

##### （1）边做边修改模型

在这种模型中，既没有规格说明，也没有经过设计，软件随着客户的需要一次又一次地不断被修改。在这个模型中，开发人员拿到项目后立即根据需求编写程序，调试通过后生成软件的第一个版本。在提供给用户使用后，如果程序出现错误，或者用户提出新的要求，开发人员重新修改代码，直到用户满意为止。这是一种类似作坊的开发方式，对编写几百行的小程序来说还可以。但这种方法对任何规模的开发来说都是不能令人满意的，其主要问题在于：

- 缺少规划和设计环节，软件的结构随着不断的修改越来越糟，导致无法继续修改。
- 忽略需求环节，给软件开发带来很大的风险。

- 没有考虑测试和程序的可维护性，没有任何文档，软件的维护十分困难。

### (2) 瀑布模型

瀑布模型将软件生命周期划分为制定计划、需求分析、软件设计、程序编写、软件测试和运行维护等六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。在瀑布模型中，软件开发的各项活动严格按照线性方式进行，当前活动接受上一项活动的工作结果，实施完成所需的工作内容。当前活动的工作结果需要进行验证，如果验证通过，则该结果作为下一项活动的输入，继续进行下一项活动，否则返回修改。

### (3) 快速原型模型

快速原型模型的第一步是建造一个快速原型，实现客户或未来的用户与系统的交互，用户或客户对原型进行评价，进一步细化待开发软件的需求，通过逐步调整原型使其满足客户的要求，开发人员可以确定客户的真正需求是什么；第二步则在第一步的基础上开发客户满意的软件产品。显然，快速原型方法可以克服瀑布模型的缺点，减少由于软件需求不明确带来的开发风险，具有显著的效果。快速原型的关键在于尽可能快速地建造出软件原型，一旦确定了客户的真正需求，所建造的原型将被丢弃。因此，原型系统的内部结构并不重要，重要的是必须迅速建立原型，随之迅速修改原型，以反映客户的需求。

### (4) 增量模型

在增量模型中，软件被作为一系列的增量构件来设计、实现、集成和测试，每一个构件是由多种相互作用的模块所形成的提供特定功能的代码片段构成。增量模型在各个阶段并不交付一个可运行的完整产品，而是交付满足客户需求的一个子集的可运行产品。整个产品被分解成若干个构件，开发人员逐个构件地交付产品，这样做的好处是软件开发可以较好地适应变化，客户可以不断地看到所开发的软件，从而降低开发风险。

### (5) 螺旋模型

1988年，Barry Boehm正式发表了软件系统开发的螺旋模型，它将瀑布模型和快速原型模型结合起来，强调了其他模型所忽视的风险分析，特别适合于开发大型复杂的系统。

螺旋模型沿着螺旋线进行若干次迭代，其中，四个象限代表了以下活动。

- 制定计划：确定软件目标，选定实施方案，弄清项目开发的限制条件。
- 风险分析：分析评估所选方案，考虑如何识别和消除风险。
- 实施工程：实施软件开发和验证。
- 客户评估：评价开发工作，提出修正建议，制定下一步计划。

### (6) 演化模型

主要针对事先不能完整定义需求的软件开发。用户可以给出待开发系统的核心需求，并且当看到核心需求实现后，能够有效地提出反馈，以支持系统的最终设计和实现。软件

开发人员根据用户的需求, 首先开发核心系统。当该核心系统投入运行后, 用户试用, 并提出精化系统、增强系统能力的需求。软件开发人员根据用户的反馈, 实施开发的迭代过程。第一迭代过程均由需求、设计、编码、测试、集成等阶段组成, 为整个系统增加一个可定义的、可管理的子集。在开发模式上采取分批循环开发的办法, 每循环开发一部分功能, 它们就成为这个产品的原型的新增功能。于是, 设计就不断地演化出新的系统。实际上, 这个模型可被看做是重复执行的多个“瀑布模型”。

每个软件开发组织应该选择适合于该组织的软件开发模型, 并且应该随着当前正在开发的特定产品特性而变化, 以减小所选模型的缺点, 充分利用其优点。表 5-1 列出了几种常见模型的优缺点。

表 5-1 几种常见软件模型的优缺点

模 型	优 点	缺 点
瀑布模型	文档驱动	系统可能不满足客户的需求
快速原型模型	关注满足客户需求	可能导致系统设计差、效率低, 难以维护
增量模型	开发早期反馈及时, 易于维护	需要开放式体系结构, 可能会设计差、效率低
螺旋模型	风险驱动	风险分析人员需要有经验且经过充分训练

## 2. 需求分析

软件产业存在的一个问题就是缺乏统一定义的名词术语来描述我们的工作。客户所定义的“需求”对开发者似乎是一个较高层次的产品概念, 而开发人员所说的“需求”对用户来说又像是详细设计了。实际上, 软件需求包含着多个层次, 不同层次的需求从不同角度和不同程度反映着细节问题。需求有两种类型: 功能需求和非功能需求。功能需求说明这个系统必须做什么, 而非功能需求说明系统的其他属性, 如物理尺寸、价格、功耗、设计时间、可靠性等。

对一个大系统进行需求分析是一项复杂而费时的的工作, 但是, 获取少量格式清晰、简单明了的信息是理解系统需求的一个良好开端。

IEEE 软件工程标准词汇表(1997 年)中将需求定义为:

- 用户解决问题或达到目标所需的条件或能力 (capability)。
- 系统或系统部件要满足合同、标准、规范或其他正式规定文档所需具有的条件能力。
- 一种反映上面两点所描述的条件或能力的文档说明。

## 3. 设计

系统设计被定义为那些用来说明一个详细的计算机系统方案的任务, 也被称之为物理设计。嵌入式系统设计分为系统架构设计、硬件子系统设计、软件子系统设计。

### (1) 系统架构设计

系统架构设计主要描述系统如何实现规格说明中定义的功能。在嵌入式系统设计时要通盘考虑系统的硬件和软件。通常的处理是先考虑系统的软件架构,然后再考虑其硬件实现。系统结构的描述必须符合功能上的和非功能上的需求,不仅所要求的功能要体现,而且对成本、速度、功耗等非功能约束也要满足。系统原始框图中的功能要素应逐个考虑和细化,把原始框图转化为软件和硬件的系统结构的同时考虑非功能约束,是一个切实可行的方法。

### (2) 硬件子系统设计

嵌入式系统的开发环境由四部分组成:目标硬件平台、嵌入式操作系统、编程语言和开发工具。其中处理器和操作系统选择应当考虑更多的因素,避免错误的决策影响项目的进度。选择处理器时应当注意:处理器的处理速度、技术指标、开发人员对处理器的熟悉程度、处理器的 I/O 功能是否满足系统的需求、处理器的相关软件支持工具、处理器的调试是否方便、处理器制造商的支持可信度。

硬件子系统设计时先将硬件划分为部件或模块,并绘制部件或模块连接框图。其次,对每个模块进行细化,把系统分为更多个可管理的小块,从而可以被单独实现。软硬件划分和软硬件接口设计时需要注意 I/O 端口、硬件寄存器、内存映射、硬件中断、存储器空间分配等方面的信息。

### (3) 软件子系统

如今在子系统设计时,根据需求分析阶段的规格说明文档,确定系统计算模型,对软件部分进行合理的设计即可。

系统设计时需要对嵌入式操作系统进行选择,在选择嵌入式操作系统时,需要做以下几个方面的考虑:操作系统的功能、配套开发工具的选择、操作系统移植难易程度、操作系统内存的需求、操作系统的支持、操作系统的实时性。

系统编程时需要对编程语言的选择,一般选择需要注意以下几点:通用性、可移植性、执行效率、可维护性、基本性能。

软件的开发过程主要有以下步骤:

- ① 选择开发语言,建立交叉开发环境。
- ② 更加详细设计说明编写源代码,进行交叉编译、连接。
- ③ 目标代码的重定位和下载。
- ④ 在宿主机或目标机调试、验证软件功能。
- ⑤ 进行代码优化。

在开发过程中,需要注意软件开发文档的制作。在嵌入式产品的开发设计过程中,开发阶段完成系统产品的实现,这一阶段同时需要完成一系列文档,这些文档对完成产品设

计、维护相当重要。这些文档分别为技术文件目录、技术任务书、技术方案报告、产品规格、技术条件、设计说明书、试验报告、总结报告等。

#### 4. 系统集成与测试

通常嵌入式系统测试主要包括软件测试、硬件测试、单元测试三个部分。

一般系统的硬件测试包括可靠性测试和电磁兼容性测试，关于电磁兼容性目前已经有了强制性国内和国际标准。

嵌入式系统软件测试方法和原理跟通用软件的测试基本一致。软件测试时，一般需要测试实例或测试序列，序列有两种来源：一种是需要用户进行设计，另一种是标准的测试序列。无论哪种测试实例，都要求实例能够大概率地发现最大的错误，但在测试的内容上有些差别。

- 嵌入式软件必须长时间稳定运行。
- 嵌入式软件一般不会频繁地进行版本升级。
- 嵌入式软件通常使用在关键性的应用中。
- 嵌入式软件必须和嵌入式硬件一起对产品故障和可靠性负责。
- 现实世界的条件是异步和不可测的，使得模拟测试非常困难。

由于这些差别，使得嵌入式系统软件测试主要集中在如下四个方面：

- 因为实时性和同时性很难同时满足，所以大多数测试集中于实时测试。
- 大多数实时系统都有资源约束，所以需要更多的性能和可用性测试。
- 可以使用专用实时跟踪工具对代码覆盖率进行测试。
- 对可靠性的测试级别比通用软件要高得多。

另外，性能测试也是设计嵌入式系统中需要完成的最主要的测试活动之一，对嵌入式系统有决定性的影响。

### 5.1.2 系统开发项目管理基础知识及其常用管理工具使用方法

“项目管理”给人的一个直观概念就是“对项目进行的管理”，这也是其最原始的概念，它有两个方面的内涵，即：项目管理属于管理的大范畴、项目管理的对象是项目。

然而，随着项目及其管理实践的发展，项目管理的内涵得到了较大的充实和发展，当今的“项目管理”已是一种新的管理方式、一门新的管理学科的代名词。可见，“项目管理”一词有两种不同的含义：其一是指一种管理活动，即一种有意识地按照项目的特点和规律，对项目进行组织管理的活动；其二是指一种管理学科，即以项目管理活动为研究对象的一门学科，它是探求项目活动科学组织管理的理论与方法。前者是一种客观实践活动，后者是前者的理论总结；前者以后者为指导，后者以前者为基础。就其本质而言，两者是统一的。基于以上观点，我们给项目管理做出以下定义。





项目管理就是以项目为对象的系统管理方法，通过一个临时性的、专门的柔性组织，对项目进行高效率的计划、组织、指导和控制，以实现项目全过程的动态管理和项目目标的协调与优化。所谓实现项目全过程的动态管理是指在项目的生命周期内，不断进行资源的配置和协调，不断做出科学决策，从而使项目执行的全过程处于最佳的运行状态，产生最佳的效果。所谓项目目标的综合协调与优化是指项目管理应综合协调好时间、费用及功能等约束性目标，在相对较短的时期内成功地达到一个特定的成果性目标。项目管理的日常活动通常是围绕项目计划、项目组织、质量管理、费用控制、进度控制等五项基本任务来展开的。项目管理贯穿于项目的整个寿命周期，它是一种运用既有规律又经济的方法对项目进行高效率和质量进行考核，并注重将当前的执行情况与前期进行比较。在典型的项目环境中，尽管一般的管理办法也适用，但管理结构需以任务（活动）定义为基础来建立，以便进行时间、费用和人力力的预算控制，并对技术、风险进行管理。在项目管理过程中，项目管理者并不对资源的调配负责，而是通过各个职能部门调配并使用资源，但最后决定什么样的资源可以调拨取决于业务领导。

一般来说，列作项目管理的一般是指技术上比较复杂、工作量比较繁重、不确定性因素很多的任务或项目。第二次世界大战期间美国对原子弹，以及后来的阿波罗计划等重大科学实验项目就是最早采用项目管理的典型例子。项目管理的组织形式在 20 世纪 50、60 代开始被广泛应用，尤其在电子、核工业、国防和航空航天等工业领域中应用更多，目前项目管理已经应用在几乎所有的工业领域中。项目管理是以项目经理（Project Manager）负责制为基础的目标管理。

对于软件的开发管理来讲，软件范围管理、软件进度管理、软件成本管理、软件配置管理、软件质量管理、软件风险管理、开发人员管理等 7 个方面的管理尤为重要，软件开发的每个阶段、每个过程都要重视这几个方面的管理。

### 1. 项目范围管理

所谓项目范围管理，包括保证项目包含所有为顺利完成项目所需的全部工作需要的过程。其目的是控制项目的全部活动都在需求范围内，以确保项目资源的高效利用。它主要包括项目启动、范围计划编制、范围定义、范围核实和范围变更控制等五个部分的内容。项目启动指批准项目启动或者允许项目进入下一个阶段；范围计划编制是将生产项目产品所需进行的项目工作渐进明细和形成文件的过程；项目范围定义是把主要的项目可交付成果分解成更小、更易管理的单元，达成如下目的：

- 提高对于成本、时间及资源估算的准确性。
- 为绩效策略与控制定义一个基准计划。
- 便于进行明确的职责分配。

正确的范围定义是项目成功的关键。范围核实是项目关系人（发起人、客户）正式接

受项目范围的过程。范围核实需要审查可交付成果和工作结果，以确保它们都已经正确圆满地完成。如果项目被提前中止，范围核实过程应当对项目完成程度建立文档。范围核实与质量控制是不同的，范围核实是对有关工作结果的“接收”，而质量控制是有关工作结果的正确性。项目范围变更控制涉及的是：

- 对造成范围变更的因素施加影响，以确保这些变更得到一致认可。
- 确定范围变更是否已经发生。
- 当范围变更发生时对实际变更进行管理。

范围变更控制必须与其他控制管理过程（进行控制、成本控制和质量控制）结合在一起使用，才能取得良好的效果。

## 2. 项目成本管理

所谓项目成本管理，是保证在批准预算内完成项目所需要的过程。成本对项目有关各方来说是非常敏感的问题，所以成本管理在软件项目管理中是一项非常重要的工作。软件项目的成本不仅包括开发成本，也包括开发之前立项阶段以及软件在运行中的费用。此外，操作者的培训费用和项目所使用的各种硬件设施费用也都是整个项目成本的一部分，这些成本都需要很好地计划和控制。

项目成本管理包括资源计划编制、成本估算、成本预算、成本控制四个主要部分。资源计划编制是确定为完成项目各活动需要什么资源（人、设备、材料）和这些资源的数量。资源计划与成本估算是紧密相关的，成本估算就是计算出完成一个项目的各活动所需各资源成本的近似值。

当一个项目按合同进行时，应区分成本估算和定价这两个不同意义的词。成本估算所涉及的是对可能数量结果的估算——执行组织为提供产品和服务的花费是多少；而定价是一个商业决策——执行组织为提供的产品或服务索取多少费用。成本估算是定价要考虑的因素之一。成本估算包括确认和考虑各种不同的成本估算替代方案，例如，软件设计阶段多做些工作可减少编码阶段的成本，而成本估算过程必须考虑增加的设计工作所多出的成本是否被以后的节省所抵消。

成本预算是把估算的总成本分配到单个活动或工作包上去，建立基准计划来度量项目实际绩效。成本控制的内容有：对造成成本基准计划变化的因素施加影响，以保证这种变化得到一致认可；确定成本基准计划是否已经发生变化；当变化发生和正在发生时，对这种变化执行管理。

## 3. 项目时间管理

时间管理包括确保项目按时完成所需的各个过程，它包括活动定义、活动排序、活动历时估算、进度计划编制、进度控制等五个部分内容。活动定义是对 WBS 中规定的可交付成果或半成品的产生所必须进行的具体活动进行定义，并形成文档。为使项目目标得以

实现, 在这个过程中, 对活动做出定义无疑是必要的。活动排序是指确定各活动之间的依赖关系, 并形成文档。活动必须正确地加以排序, 以便今后制定切实可行的进度计划。排序可由计算机辅助或用手工完成。

项目活动历时估算是根据项目范围和资源的相关信息为进度表设定历时输入的过程。历时估算的输入通常来自项目团队中熟悉该活动特性的个人和团体。估算通常采用渐进明细的方式, 同时此过程需要考虑输入数据的质量和可获得性。因此, 可以假设此估算逐步精确, 并且其质量水平是已知的。项目团队中最熟悉具体活动性质的个人或团队应当完成历时估算。制定进度计划要确定项目活动的开始和结束日期, 若开始和结束日期是不现实的, 项目就不可能按计划完成。进度计划、历时估算、成本估算等过程交织在一起, 这些过程反复多次, 最后才能确定项目进度计划。进度控制涉及的情况是:

- 对造成进度变更的因素影响, 以确保这些变更得到一致认可。
- 确定进度变更是否已经发生。
- 当变更发生时对实际变更进行管理。

#### 4. 配置管理

随着软件规模和复杂性的增大, 软件开发的预算和准时发布是许多软件企业经常回避的问题。许多大型开发项目往往都会延迟和超出预算, 软件开发不得不直面越来越多的问题, 表现为开发的环境日益复杂, 代码共享日益困难, 需跨越的平台增多; 软件的重用性需要提高; 软件的维护越来越困难。

为了解决这些问题, 软件配置管理 (Software Configuration Management, SCM) 作为控制软件系统变化的学科应运而生, 其主要作用是通过结构化的、有序化的、产品化的管理软件工程的方法来维护产品的历史, 鉴别和定位产品独有的版本, 并在产品的开发和发布阶段控制变化。通过有序管理和减少重复性工作, 配置管理保证了生产的质量和效率。它涵盖了软件生命周期所有领域并影响所有数据和过程。作为软件开发中一个重要过程, 实现在有限的时间和资金内, 满足不断增长的软件产品质量要求, 软件配置管理已经逐渐受到各软件企业的重视。

对于软件配置管理, IEEE 给出了一个定义: SCM 是指在软件系统中确定和定义构件 (源代码、可执行程序、文档等), 在整个生命周期中控制发布和变更, 记录和报告构件的状态和变更请求, 并定义完整的、正确的系统构件的过程。在 IEEE 标准: 729—1983 中, 软件配置管理包括以下功能。

- 配置标识: 产品的结构、产品的构件及其类型。为其分配唯一的标识符, 并以某种形式对它们进行存储。
- 版本控制: 通过建立产品基线, 控制软件产品的发布和在整个软件生命周期中对软件产品的修改。

- 状态统计：记录并报告构件和修改请求的状态，并收集关于产品构件的重要统计信息。
- 审计和审查：确认产品的完整性并维护构件间的一致性，即确保产品是一个严格定义的构件集合。
- 生产：对产品的生产进行优化管理。它将解决最新发布的产品应由哪些版本的文件和工具来生成的问题。
- 过程管理：确保软件组织的规程、方针和软件周期得以正确贯彻执行。它将解决要交付给用户的产品是否经过测试和质量检查的问题。
- 小组协作：控制开发统一产品的多个开发人员之间的协作。例如，它将解决是否所有本地程序员所做的修改都已被加入到新版本的产品中的问题。

### 5. 软件配置管理的解决方案

并发版本系统 (Concurrent Versions System, CVS) 是主流的开放源码的版本控制系统，是 Linux 和 UNIX 下系统自带的版本控制工具。CVS 对于从个人开发者到大型、分布式团队都是有用的。CVS 提供了基本的认证安全和版本控制机制。版本控制是工作组软件开发中的重要方面，它能防止意外的文件丢失，允许反追踪到早期版本，并能对版本进行分支、合并和管理。在软件开发中比较两种版本的文件或找回早期版本的文件时，源代码的控制是非常有用的。CVS 是一种源代码控制系统，它提供了完善的版本和配置管理功能，以及安全保护和跟踪检查功能。CVS 带有一个专业的文档、代码管理库，通过将有关项目文档（包括文本文件、图像文件、二进制文件）存入数据库进行项目研发管理工作。CVS 能够对文件进行控制，用户可以根据需要随时快速有效地共享文件，从文档的控制流程（增加、删除、修改、借阅等），到文档的修改信息记录，实现完善的文档管理。CVS 提供历史版本的提取、提供源码历史版本对比。CVS 文件一旦被添加进 CVS，它的每次改动都会记录下来，用户可以恢复文件的早期版本，项目组的其他成员也可以看到有关文档的最新版本，并对它们进行修改，CVS 也同样会将新的改动记录下来。还会发现，用 CVS 来组织管理项目，使得项目组间的沟通与合作更简易而且直观。

## 5.1.3 系统开发工具与环境知识

### 1. 建模工具

简单地说，“建模”就是建立软件系统抽象模型。系统模型贯穿软件生命周期的整个过程，包括分析模型、设计模型、实现模型、测试模型等，但通常所说的“系统模型”主要指分析模型和设计模型。

面向对象的方法学，把系统看做是相互协作的对象。这些对象是结构和行为的封装，都属于某个类，那些类具有某种层次化的结构，系统的所有功能通过对象之间相互发送消

息来获得。面向对象的系统模型有着十分明显的优点：抽象化、封装化、层次化、分类、并行、稳定、可重用、可扩展。

统一建模语言UML是可视化建模语言中的一种，属于第三代面向对象建模语言。它将模型中的信息用标准图形元素直观地表示出来，使用户、开发人员、设计人员、测试人员、管理人员和其他涉及项目的人员可以更容易地交流。最常用的可视建模语言有Booch法、对象建模技术(OMT)和统一建模语言UML。其中，UML是ANSI和OMG组织所采用的标准，被世界上绝大多数公司所接受。

UML是由世界著名的面向对象技术专家Grady Booch、Jim Rumbaugh和Ivar Jacobson发起，在著名的Booch方法、OMT方法和OOSE方法的基础上，集众家之长，几经修改而完成的，适用于系统开发的不同阶段。采用UML进行设计具有以下特点和优势：

- UML语言简单、易学、易用。
- UML采用可视化的图形描述，比较形象直观。
- 可以使不同技术背景的开发人员和设计人员更容易地相互交流。
- UML语言是第三代面向对象建模语言的标准，被绝大多数业内人士认同。
- UML采用图形化的设计，将系统的核心部分描述出来，可以供以后系统开发使用。
- 有利于项目的回溯和测试。

UML是面向对象技术发展的重要成果，获得了科技界、工业界和应用界的广泛支持，已成为可视化建模语言事实上的工业标准。本书介绍的“建模工具”仅包括以UML为建模语言的分析设计模型生成工具。

目前，在UML建模领域，已经有很多厂商在角逐，知名的UML建模工具已有一百多种，每家都宣称自己是最好的，最合适用户的需求。那么，什么才是选择建模工具的客观判断标准呢？UML建模专家提出了建模工具应该具有的八条特性：

- 全面支持UML。
- 能自动保持源码和模型的同步，无须人工干预。
- 具有强大的文档生成能力。
- 能与软件工程领域的其他工具进行集成。
- 能支持团队工作。
- 支持设计模式。
- 支持重构。
- 具有反向工程能力。

下面介绍几种典型的建模工具：

#### (1) Rhapsody 软件介绍

I-Logix公司是目前唯一解决了在实时系统中完全运用UML语言的公司，其相关产品

Rhapsody 在业界享有盛誉。美国 NASA 的火星探路者航天器就是运用 Rhapsody 在 Vx Works 上开发应用程序的。

Rhapsody 是一个基于 UML 的面向嵌入式实时应用开发的集成、可视化环境。软件开发者可以在这个环境里进行分析、设计、实现及验证。系统分析和设计用 UML 来描述，对系统建模；实现过程利用代码自动生成技术来实现；测试过程将依赖于生成的代码，通过在代码中拆装一些用于支持模型调试的调试信息来实现；而代码的编译、连接则采用目标系统的操作系统开发环境来完成；代码的运行与源程序级的调试仍然采用一般的嵌入式软件调试环境。图 5-1 所示为支持基于 UML 的迭代式开发方法的开发环境的结构，虚框部分为基于 UML 的软件开发环境。

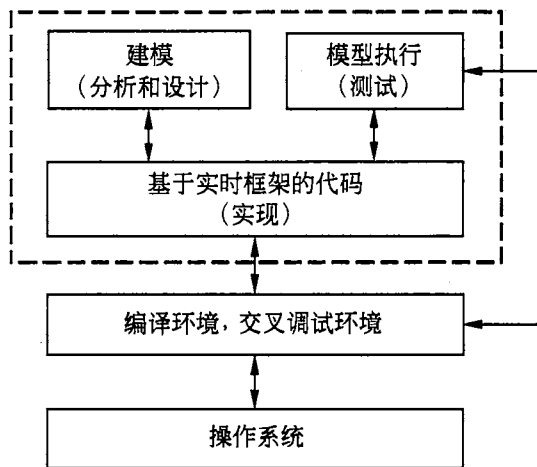


图 5-1 基于 UML 嵌入式软件开发环境的结构

Rhapsody 的主要特点：

- 实时嵌入式软件的可视化开发，遵从 UML。
- 在早期阶段对软件的设计进行调试、验证和优化。
- 生成 100%精确高质量 C、C++或 Java 代码。
- 设计与代码实现紧密相关。
- 针对实施和分布式应用。
- 提供完整的设计文档（可生成中文文档）。
- 提升软件重用能力。
- 提升软件集成的效率。
- 提倡迭代式开发，提升团队开发的效率。
- 减少技术人员跳槽的影响。





- 缩短软件开发周期至少 30%。
- 它是基于传统嵌入式软件开发环境之上的高层开发环境，基于模型的设计和调试是它的侧重点，同时借助于传统的软件调试环境可以实现基于模型的调试和基于源程序的调试同步进行。
- 模型的运行和调试是基于与最终产品的源程序相同或相近的程序来实现的，这种模型级的调试与其他的模型仿真本质不同。
- 程序的生成是基于实时框架的，使应用软件的开发与具体的软件平台无关，对于不同的操作系统平台只需要维护不同的实时框架即可。

Rhapsody 是业界唯一的完全基于 UML 的实时应用软件开发环境，它使需求分析、软件设计、代码实现和功能测试完美地融为一体，实现了软件开发自动化。Rhapsody 支持基于模型的调试；提供专门为实施嵌入式应用软件的可执行的框架，可以产生基于 VxWorks、POS、OSE 等多种操作系统的 C 语言、C++ 语言、Java 语言的源程序。使用 Rhapsody 可以将工作重点从编码转移到设计，有利于团队开发；可以改善产品质量，缩短开发周期。

### (2) IBM Rational Rose

IBM Rational Rose 在软件工程领域被公认为是 UML 建模工具的执牛耳者。Rose 为大型软件工程提供了可塑性和柔性极强的解决方案：

- 强有力的浏览器，用于查看模型和查找可重用的组件。
- 可定制的目标库或编码指南的代码生成机制。
- 既支持目标语言中的标准类型又支持用户自定义的数据类型。
- 保证模型与代码之间转化的一致性。
- 通过 OLE 链接，Rational Rose 图表可动态链接到 Microsoft 中。
- 能够与 Rational Visual Test、SQA Suite 和 SoDA 文档工具无缝集成，完成软件生命周期中全部辅助软件工程师工作。
- 强有力的正/反向建模工作。
- 缩短开发周期。
- 降低维护成本。

### (3) IAR VisualState

VisualState 是一组高级嵌入式设计工具套件，专门用于开发高质量的嵌入式软件。VisualState 提供了软件开发者一个高级的层次化的系统分析方法，使软件开发者可以从系统全局出发，对一个复杂应用的结构一目了然，从而简化了系统设计和维护。

VisualState 是一个集成化的软件开发工具包，特别适用于开发处理复杂任务的嵌入式系统。VisualState 的软件涵盖了以下五个嵌入式产品的开发步骤，每个工具模块都有最先进的图形化功能支持：

- 图形化的设计。
- 快速的原型生成。
- 自动代码生成。
- 广泛的测试。
- 自动的开发文档生成。

VisualState 是一套集成化的图形化的工具链，能支持从最初设计到最终产品代码集成的整个嵌入式开发周期。VisualState 工具链支持以下功能：

- 状态机的设计
- 原型设计
- 代码生成
- 产品集成
- 测试
- 文档生成

## 2. 编程工具

编程工具是指辅助编程过程活动的各类软件。从方法学分类，可分为结构化编程工具和面向对象的编程工具；从使用方式分类，可分为批处理式编程工具和可视化编程工具；从功能分类，可细分为编辑工具、编译（汇编）工具、组装工具和排错工具等，但目前的编程过程多采用集成化开发环境工具。

下面介绍两种编程工具。

### （1）集成开发环境 Tornado

Tornado 2.2 是嵌入式实时领域里最新一代的开发调试环境，它给嵌入式系统开发人员提供了一个不受目标机资源限制的超级开发和调试环境。Tornado 包含三个高度集成的部分：

- 运行在宿主机和目标机上的强有力的交叉开发工具和实用程序。
- 运行在目标机上的高性能、可剪裁的实时操作系统 VxWorks。
- 连接宿主机和目标机的多种通信方式，如以太网、串口线、ICE 或 ROM 仿真器等。

在集成开发环境方面，VxWorks Tornado II 提供了二次开发途径。集成开发环境是一种更直观的自动化环境，使得使用经验不同的开发人员可以快速、方便地在 VxWorks 上面开发应用，甚至使开发工作成为一种乐趣。

所有的开发工具都适用于开发不同类型的目标机，Tornado 是专门为解决嵌入式开发人员所面临的诸多问题而设计的。对于不同的目标机，Tornado 给开发者提供一个一致的图形接口和人机界面。当使用 Tornado 的开发人员转向新的目标机时，不必再花费时间学习或适应新的工具；对深嵌入式应用开发者来说更重要的是，Tornado 所有的工具都是驻留在开发平台上的。在嵌入式系统工具发展历史上，Tornado 是第一个实现了当目标机资



源有限时开发工具仍可使用而且功能齐全的开发环境。另外，所有工具都通过一个中央服务器（target server）处理与目标机的通信。因此，无论连接方式是 Ethernet，还是串口线、ICE 仿真器、ROM 仿真器或客户设计的调试通道，所有工具均可使用。

Tornado 工具集包括以下各项。

- **Launcher:** Tornado 的框架环境、管理目标机与工具。
- **Browser:** 详细查看任何目标系统对象、内存、堆栈和 CPU 占用率。
- **WindConfigTM:** 图形化自动配置工具。
- **CrossWind:** 系统和任务级调试工具，可以调试 C、C++ 以及汇编程序。
- 基于 Internet 的技术支持工具。
- 适用于所有目标机结构的编译器。
- 动态下载器：对象模块的动态链接和加载及卸载。
- **WindShTM:** 与目标机进行交互的命令（C、TCL 语言）解释工具。
- 支持 C 和 C++。
- 开放的、可扩展的开发环境。

除了提供适用于不同目标机的工具集以外，Tornado 还是一个完全开放的环境，开发人员或第三方厂商可以很容易地把自己的工具集成到 Tornado 框架下。这种开放的环境使得开发人员可以使用各种各样而且越来越多的第三方软件及硬件工具，从而进一步提高开发人员的工作效率。

## （2）Source Insight 编辑器

Source Insight 是一个功能强劲的程序编辑器，它内置对 C/C++、Java 和 x86 汇编语言程序的解析；有自己的动态数据库，在编程时可提供有用的文本提示，包括有关的函数、宏、参数等。

Source Insight 是一个面向项目的、支持多种开发语言（Java、C、C++等）的编辑器和浏览器，它可运行在 Linux、Windows 操作系统上。Source Insight 还支持查找、定位、彩色显示等功能。

当编辑程序时，Source Insight 可以：

- 更新最新的日期信息。
- 动态类型解析。
- 字符窗口。
- 动态上下文窗口。鼠标指针所指的字符或正在输入的字符，Source Insight 会自动显示字符的定义。
- 句法格式化。
- 混合语言编辑。

- 符号自动完成。
- 所用符号和文件的快速访问。
- 编程项目支持。
- 团队项目支持。
- 参考功能。搜索时在整个工程中找到所有的标记，并且在该程序的前面加上红色箭头。
- 源码链接。
- 快速项目文件搜索和替代。
- 拖放编辑。
- 粘贴窗口。
- 同外部编译器和工具集成。

### 3. 测试工具

由于嵌入式软件的特殊性，很难用一般的软件测试工具对它进行测试。人们一直在致力于寻找适合测试嵌入式软件的方法和工具，尤其是在进入 20 世纪 90 年代后，随着嵌入式软件在军用和民用高科技项目中的广泛使用，嵌入式软件的质量问题也不断涌现。为了解决这些质量问题，人们开始投入更多的物力和人力进行研究。针对嵌入式软件的特点，人们提出了多种嵌入式软件的测试方法。

#### (1) 白盒侵入式检测工具

白盒测试范畴的侵入式检测工具主要是对测试软件进行软件性能、代码覆盖率、内存分配的测试、分析。

属于典型的侵入式检测工具有以下几种。

##### ① LynxInsure++

用于检测代码、目标系统可执行代码纠错以及提供内存泄露探测和覆盖测试分析。目前只能应用于 LynxOS 系统的开发，包括 4 个部分。

- Insure++：同标准 lynx 编译器一起运行的源码检测工具。可检查初级错误、API 应用中的类型和参数错误、指针和数组错误、字符串操作错误。
- Inuse：执行于目标系统的内存检测工具。可查找内存漏洞、检查动态内存，减少碎片。
- TCA：程序的覆盖检测。可提供完全的覆盖报告，检测因为块和函数引起的断裂。
- Multi Run-Time Error Checker：通过向程序插入特殊代码来查看、报告实时系统的错误。它检查的错误包括：
  - ◆ 读写未分配的内存。
  - ◆ 释放未分配内存。

- ◆ 未被任何程序指针应用的动态内存分配块。
- ◆ 超出定义范围的数组元素。
- ◆ 向变量或内存区存储超其容量大小的值。
- ◆ 用空指针访问内存。
- ◆ 用参数决定程序分支时，出现参数值没有对应分支的情况。
- ◆ 定义从未用过的局部变量。
- ◆ 访问指定内存地址错误。
- ◆ 从一个没有任何反馈的函数中退出。

## ② CodeTest

作为全球第一台专为嵌入式系统软件测试而设计的工具套件，CodeTest 为追踪嵌入式应用程序、分析软件性能、测试软件的覆盖率以及存储体的动态分配等提供了一个实时在线的高效率解决方案。CodeTest 包括三个产品（分别用于嵌入式软件系统开发的不同阶段的测试）。

- CodeTest Native：在主机上完成软件开发后的测试。
- CodeTest Software-In-Circuit：将软件植入目标系统，通过以太网连接进行软件测试。
- CodeTest Hardware-In-CircuitT：系统测试，如系统性能、产品质量等，需要软硬件配合测试。

## (2) 黑盒

属于黑盒测试的方法目前有实物测试、全数字仿真测试，以及利用仿真测试环境进行测试。

### ① 实物测试

实物测试就是运行实际的系统，通过考核软件的运行情况来达到对软件的测试目的，如飞机试飞、生产线试运行都属于实物测试范畴。

实物测试的优点是：测试环境真实可靠，因为测试环境就是运行环境；错误疑点确定为错误的概率很高。

但是与它的优点相比，实物测试的缺点似乎更引人注目：

- 测试环境难于构建。
- 测试费用昂贵。
- 测试结果难以评价。
- 测试过程难以复现。
- 有时需要额外的设备。

虽然一般的嵌入式系统都需要做实物测试，但是由于上述实物测试的种种缺点，人们

应争取尽量减少实物测试的时间。

### ② 全数字仿真测试

全数字仿真测试就是将被测代码放置在数学平台上运行测试，数学平台仿真了被 4% 软件运行所需的全部环境。很多嵌入式软件的开发工具提供的仿真器可以被认为是一种数学平台，它们是在主机上模拟目标机的运行环境，供开发者调试、测试嵌入式软件。

VectorCast 也是这种工具之一，它可以测试 Ada、C/C++ 和嵌入式 C++ 源代码，自动生成测试代码来为主机和嵌入式环境构造可执行的测试架构。使用 VectorCast 测试系统，可以方便地建造起一个独立单个软件部件所需的测试环境。它还提供构造和运行测试范例，并生成关于实际结果与预测结果之间的统计信息所需报告的部件。

全数字仿真测试方法的优点如下：

- 将被测代码与它的真实运行环境完全分离，测试时不需要使用被测系统的硬件。
- 一旦发现被测软件错误，很容易将其定位。

但是这种方法的缺点也比较多：

- 构建数学平台的难度很大，尤其当系统复杂时。
- 建立数学平台成本很高。
- 与真实运行环境相差较大。数学平台上的模拟运行大多不能真实地反映软件的实际运行情况。
- 错误疑点确定为错误的概率的低。

虽然全数字仿真测试目前还不尽如人意，但是随着仿真与建模技术的发展，嵌入式系统开发的逐步规范化和系统化，全数字仿真测试技术的前景还是一片光明的。

### ③ 仿真测试环境

仿真测试对被测软件的测试是非侵入式的，并且无须测试者了解被测软件的代码，被测软件运行在真实的计算机系统中，测试结果真实可靠。相对于实物测试的方法，它有如下的优点：

- 在大多数情况下，采用嵌入式系统的真实运行环境来对嵌入式软件进行测试的代价都是很大的，尤其是嵌入式系统的真实运行环境包含有昂贵的易损、易耗器件时，或当软件的正确率很低时，实物测试都不可行。
- 无法获得被测系统的相关交联系统时，必须用仿真环境对嵌入式软件进行测试。
- 仿真环境能测试在罕见的特殊条件下嵌入式软件的行为。使用真实环境去再现这些特殊事件，要么特别困难，要么极其危险。

一般而言，测试人员对仿真环境的可控性要强于真实环境，在仿真环境下，被测软件的运行更加容易，也更灵活，易于评估被测系统的行为特性，也更容易获取被测系统的响应和中间结果。

相对于数字平台来说，它有如下的优点：

- 仿真测试环境实现起来相对简单，实现成本可以接受。
- 测试效果好。由于被测试软件运行于真实的环境之中，故测试结果可信度高。
- 仿真环境测试易于通用化。
- 不但能发现软件错误，并且还能发现相关硬件错误。

正是由于采用仿真测试环境进行测试有如此多的优点，嵌入式软件仿真测试环境的研究才越来越受到人们的重视。

下面是两种典型的集中式的测试工具（或平台）。

- **MessageMaster**: 测试嵌入式系统时，认为被测试系统（SUT）是一个黑匣子，要求测试人员通过向 SUT 发送消息、检查输出消息并同期望值比较来模拟真实环境。MessageMaster 提供工具，为 SUT 快速建立一个模拟环境。
- **TestQuest Pro**: 运行时像一个“虚拟用户”，模拟测试人员的工作，一步步检验被测系统，同时自动记录测试结果。TestQuest Pro 通过提供一系列模拟信号输入来导引对被测系统的测试过程，包括键盘操作、鼠标或其他输入设备所产生的电信号。为了检验设备输出，TestQuest Pro 自动捕获图形信号和通信数据，并同参考数据相比较。对被测系统的连接模拟通过可装卸模块来提供。TestQuest Pro 提供模拟输入的模块覆盖 Pointing Devices、Touch Screen、Keyboards、Keypads、Serial Comm、Discretes、USB Devices、IR Devices 等功能，提供 VGA Capture、Digital LCD Capture、S-Video Capture、Serial Comm、Discretes 等监视能力。TestQuest Pro 还提供一个开发环境，用来进行测试脚本的开发、运行和调试，并提供测试信息管理系统将准备好的测试脚本加入测试套件中执行。

近年来，随着嵌入式计算机技术的不断发展，嵌入式系统的规模在不断扩大，复杂程度也在不断增加，另外，加之分布式技术、仿真技术的高速发展，分布式仿真测试环境已经成为研究的热点。分布式，即多台计算机互相配合、协同处理，可提高测试系统的性能，提高测试系统的通用性和可扩展性，使之能够用于复杂的、实时性强的被测系统的测试。以下是几种典型的系统。

- **B-TREE 的 ValidorGold 解决方案**，它的测试环境由 Host、Sentri 作为基本系统，根据需要可以增加 Capture 节点。
- **Tech S.A.**是目前为止针对较为复杂的系统而推出的仿真测试环境中最为成功的解决方案。ADS-2 的基本系统由一台 UNIX 工作站的主控节点和 VME RTI/O System 的激励节点组成，它们之间通过 TCP/IP、共享内存或 Scramnet 的方式连接。ADS-2 支持众多的接口，具有强大的仿真功能，较好地实现了分布式构型，主要用于嵌入式系统的快速原型建立、开发、系统集成、测试和验证。虽然 ADS-2 是一个比

较优秀的仿真测试平台，但是它仍有一些缺点，如不能支持周期小于 2ms 的实时系统的测试；对某些强实时系统无法测试，并且价格昂贵；在针对复杂的系统采用分布式构型进行测试时，需要多套 ADS-2，测试成本不菲。

## 5.2 系统分析基础知识

### 5.2.1 系统分析的目的和任务

系统分析的对象主要是软件需求，所以我们应该首先了解各种需求的概念以及性质。

软件工程师所需解决的问题往往十分复杂。了解问题的性质可能是非常困难的，尤其当系统是全新的时候。因此，搞清系统应该做什么也是困难的。对系统应提供的服务和所受到的约束的描述就是系统需求关心的内容，对服务和约束的发现、分析、建立文档、检验的过程叫做需求工程。

在某些情况下，需求被视为对系统应该提供的服务或对系统约束的高层抽象描述，在一些情况下，它又被定义为是对系统功能的详细的、用数学方法的形式化描述。

需求工程过程中出现的一些问题就是因为没有对这两个层次的描述做出清晰的分离。本书采用用户需求表示高层的概要需求，采用系统需求表示对系统应该提供哪些服务进行详细描述。与这两个层次描述相对应的是软件设计描述，来连接需求工程和设计活动。用户需求、系统需求和软件设计描述的定义如下。

- 用户需求：是用自然语言加图表的形式给出的关于系统需要提供哪些服务以及系统操作受到哪些约束的声明。
- 系统需求：详细地给出系统将要提供的服务以及系统受到的约束。系统需求文档有时称为功能描述，它可能成为系统买方和软件开发者之间合同的重要内容。同时它也是系统分析的主要对象。
- 软件设计描述：是对软件设计活动的概要描述，是详细设计和实现的基础。这个描述是在系统需求描述的基础上再加入更详细的内容构成的。

需要向不同类型的读者传达信息，因此，需要多个层次的系统描述。表 5-2 说明了用户需求和系统需求之间的区别，它说明用户需求是如何扩展为一系列系统需求的。只有了解了这一转变过程，才能更好地对系统进行分析。

用户需求是为客户和承包商管理者编写的，他们一般不具备具体技术细节方面的知识，如图 5-2 所示。系统需求是为高级技术人员和项目管理者编写的，这些技术人员既包括客户方的，也包括承包商方的。系统最终用户可能两个文档都要读。最后，软件设计描述是一个面向实现的文档，它是为具体开发系统的软件工程师编写的。

表 5-2 用户需求和系统需求

用户需求定义
1. 软件必须提供表达和访问外部文件的手段，这些外部文件是由其他工具创造的
系统需求描述
1.1 为用户提供定义外部文件类型的工具
1.2 每种外部文件类型具有一个相关联的工具
1.3 每种外部文件类型在界面上用一种专门的图标来表示
1.4 提供一种工具，使用图标表示由用户定义的外部文件类型
1.5 当用户选择一个代表外部文件的图标时，选择的效果是将与该外部文件类型相关联的工具启动起来

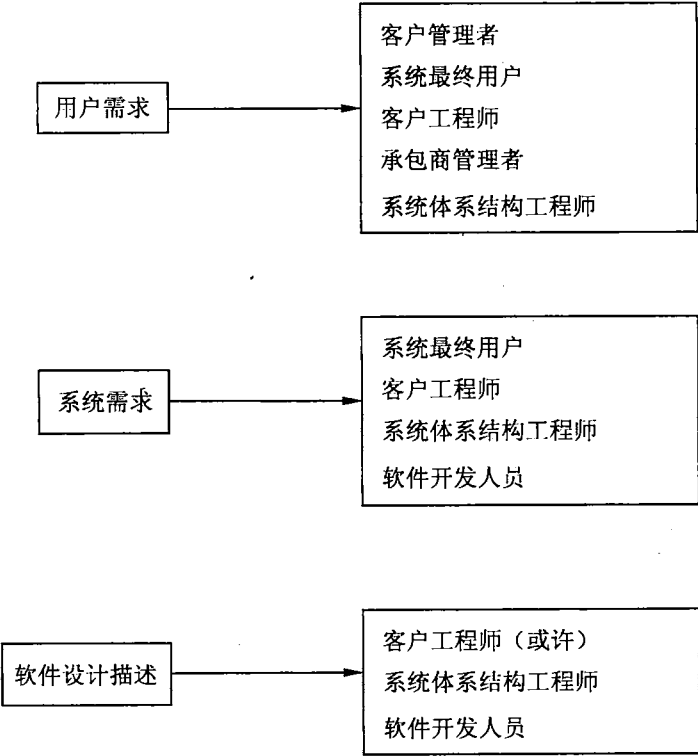


图 5-2 不同类型描述的读者对象

5.2.2 用户需求

用户需求是从用户角度来描述系统功能和非系统需求，以便让不具备专业技术方面知

识的用户能看懂。用户需求描述只描述系统的外部行为，要尽量避免对系统设计特性的描述。因而，用户需求不能使用任何实现模型来描述，而是用自然语言、图表和直观的图形来叙述。

表 5-3 所示是一个关于 Ada 编程环境的需求实例，该实例能说明上面提到的一些问题。

表 5-3 关于编程环境数据库的需求

数据库应该支持配置对象的生成和控制，也就是说，数据库中对象本身是由其他的对象组合而成的。 配置控制工具应该允许对一个版本集中的对象根据其不完全的名字访问它们
---

这个需求包括概念上和细节层次两方面的信息。在概念上，它描述了应该提供一个配置，工具作为数据库的一个固定组成部分。在细节层次上，它描述了这些配置工具应该在无须给出全名的情况下允许对版本集中的对象进行访问。这个细节层次的描述最好是放到系统需求中描述。

在需求文档中，比较好的做法是将用户需求和细节层次需求描述分开表述。否则，用户需求的非技术类读者就可能被一些技术细节所困惑，而他们真正想看的只是一些概念性的内容，表 5-4 说明了这种情况。这个例子是从一个用于编辑软件设计模型的 CASE 工具的需求文档中选出的，用户需要在窗口中显示栅格以便能对图中实体精确定位。

表 5-4 对一个带有栅格的编辑器的用户需求

栅格工具 为协助在一个图表中定位实体，用户可通过面板上的选择项打开一个以公分或英寸为单位的栅格。最初，栅格处于关闭状态。在编辑期间，可以随时切换栅格的打开和关闭，并可以随时在英寸和公分单位之间转换。栅格选择项以调节方式提供，当图形较小时栅格线的数目将会减少，以避免图表被栅格填满
---

第一句混合了三个不同类型的需求。

- 一个概念上的功能需求描述了该编辑系统应该提供一个栅格。
- 一个非功能需求给出了关于栅格单元的详细信息（公分或英寸）。
- 一个非功能的用户界面需求定义了用户如何打开和关闭该栅格。

表 5-4 中的需求还给出了一些初始化的信息，它定义栅格最初是处于关闭状态。然而，当把它打开的时候，并未定义它使用的单位。该表还提供了一些详细的信息，如用户可以在单位之间进行转换，但并不改变栅格线之间的间距。

当用户需求包括太多信息的时候，它限制了系统开发者解决问题的创意且使需求难以理解。用户需求应该集中在需要提供的主要服务上。表 5-5 所示还是前面那个例子，不同的是重写了这个关于编辑器栅格的需求描述。



表 5-5 编辑器栅格工具的定义

**栅格工具** 编辑器应该提供栅格工具，包含一个由水平和垂直线阵列作为编辑器窗口的背景。这个栅格是被动的，即实体对准的是用户的任务基本原理：一个栅格帮助用户产生一个具有一定间隔的整齐的图表。虽然一个主动栅格能自动将实体与某栅格线对齐，但这种定位也是不精确的。用户是决定实体应该放在哪里的最好的决定者

关于需求的基本原理是很重要的，它帮助系统开发者和系统维护人员了解为什么要包括这个需求以及在需求变更时能评估其影响。举例来说，在表 5-5 中，基本原理指出一个活动栅格对被摆放的对象自动捕捉栅格线是有用的。然而，要让用户更自主地操作实体，这个做法不被采纳。显然，如果后期发生变动，采用被动栅格较主动栅格更好。

一个更特别的用户需求例子在表 5-6 中给出，该例也是定义编辑系统中的一部分，这是一个关于功能的比较详细的描述。在这种情况下，定义包括一个用户动作的列表，对于保持系统功能的一致性是有必要的。实现细节不应该包含在这个附加的信息中。因此，定义没有给出光标和符号是如何移动的，也没有给出类型是如何选择的。

表 5-6 节点创建的用户需求

#### 在设计中添加节点

1. 编辑器必须为用户提供工具，把一个特别类型的节点加入到设计中
2. 应该按下列步骤增加一个节点：

- ① 用户选择要添加的节点的类型
- ② 用户将光标移动到图形中要添加节点的恰当位置，然后指出要在该处添加节点
- ③ 然后用户将节点符号拖放到它上面指定的位置

基本原理：用户是决定该在哪里放置节点的最佳人选。这个方法对用户选择节点类型和定位提供了直接控制

### 5.2.3 系统需求

系统需求是比用户需求更详细的需求描述，也是系统分析中最重要的一个环节，是系统实现的基本依据。因此，一个完全的和一致的系统描述，是软件工程人员系统设计的起点。系统需求描述可能包括许多不同模型，如对象模型和数据流模型。原则上讲，系统需求应该陈述系统应该做什么而不包括系统应该如何实现。然而，要在细节层次上给出系统完善的定义，不提到任何设计信息事实上是不可能的。这是因为：

- 首先要给出系统初始的体系结构，借助这个框架来构造需求描述。系统需求依照构成系统的不同子系统结构来给出。
- 在大部分情况下，系统和其他已存在的系统存在互相操作。这就约束了系统的设计，同时这些约束又构成了新系统的需求。

- 使用特别的设计（例如，某版本编程环境有助于提高系统的可靠性）是系统的一个外部需求。

进行用户以及系统需求的分析是所有项目成功的基础。在软件开发中，首先对系统进行全面有效的分析可以让工程师更加明确工作的内容，所以是非常有必要的。

自然语言常用来编写系统需求描述，然后用自然语言来做更详细的描述时，深层次的问题就暴露出来了，这些问题包括：

- 自然语言的理解依赖于读者和作者对同一个词语有一致的解释。因为自然语言存在二义性，所以会造成语意理解的偏差。
- 一个自然语言书写的需求描述随意性太大，能以完全不同的方式描述相同的事物。用户很难判断什么时候需求是一样的，什么时候需求是不一样的。
- 不存在一个简单的方法，使自然语言书写的需求模块化。这种形式描述的需求极难发现相关性，要想发现变化的相关性就只能是一个个需求的分析，而不能通过对一组需求的分析很快得出结论。

因为这些问题，用自然语言书写的需求描述容易引起误会。这种误会可能在软件过程的后期才发现，而这时再去解决问题费用就相当高了。

因此，本章介绍了几种替代自然语言描述的系统分析方法，这些方法在描述中加进了一些结构以减少二义性。

### （1）结构化系统分析

结构化的系统分析是书写系统需求时对自然语言所做的严格的格式。这个方法的好处是它保持了自然语言中的绝大部分好的性质，包括表现能力和易懂性，同时又在不同程度上对描述做了一致性的约束。结构化语言对使用的词汇进行了限制，而且可以使用模板来定义系统需求。结构化语言综合了源程序语言的控制结构和图形化的突出显示方法来划分系统描述。

为了用基于格式的方法来描述系统需求，必须定义标准格式或模板来表达需求。需求描述的结构化是围绕三个主要内容进行的：一是系统操作的对象，二是系统运行的功能，三是系统处理的事件。这样一个以格式为基础的描述例子在表 5-7 中表示出。

表 5-7 使用标准格式的系统需求描述

功能	添加节点
描述	添加节点到一个已存在的设计中。用户选择节点类型和位置。添加完后，节点就是当前的选择。用户通过移动鼠标指针到要添加的位置来确定节点的位置
输入	节点类型、节点位置和设计标识符



续表

功能	添加节点
来源	节点类型和节点位置由用户输入，设计标识符来源于数据库
输出	设计标识符
目的地	设计数据库。完成后，该设计就被送入设计数据库
要求	设计图以输入设计标识符为根节点
前条件	设计处于打开状态，显示在用户屏幕上
后条件	除了在相应位置添加了一个节点外，设计没有改变
副作用	没有

当一个标准格式描述功能需求时，下列各项信息应该被包括在内：

- 实体或功能描述。
- 输入及输入来源描述。
- 输出及输出去向描述。
- 其他被引用的实体的索。
- 如果一个功能性方法被用到，前条件设定在何逻辑子句为真时执行该功能；后条件设定该功能执行之后何逻辑子句应该为真。
- 对操作的副作用（如果有的话）的描述。

使用格式化的系统分析除去了自然语言描述中的一些问题，这时在描述中减少了可变性。然而，在描述中还存在一些二义性，另外的一些表示方法使用更多的结构化符号来解决这个问题，如下面要介绍的 PDL 方法。但是，非专业人员往往觉得太难掌握。

## （2）使用 PDL 的系统分析

为了解决自然语言描述固有的二义性问题，一个可行的方法是使用程序描述语言来描述需求，这样的语言称为 PDL。PDL 起源于像 Java 或 Ada 这样的程序设计语言，它包含附加的、更抽象的构造来提高它的表达能力。使用 PDL 的好处是它可以用软件工具对其进行语法和语义检查，需求遗漏和不一致也可以通过这些检查来发现。

使用 PDL 语言能得到非常详细的需求描述，有时，它们与需求文档中的内容已经相当接近。然而，我们还是推荐在两种情况下使用 PDL 语言：

- 当操作能分解为一个比较简单的动作序列并且执行顺序非常关键的时候。这样的顺序描述在自然语言中有时是紊乱的，特别是在有内嵌条件和循环的时候。这在表 5-8 中举例说明，这是一个对自动柜员机(ATM)的描述部分，用 Java 作为 PDL，但是为节省空间故意留下部分描述未列。
- 当硬件和软件的接口已经被定义了的时候。在许多情况下，子系统之间的接口在系统需求描述中已被定义，使用 PDL 可以定义接口对象和类型。

表 5-8 ATM 操作的 PDL 描述

---

```

class ATM {
    // declarations here
    public static void main (String args[]) throws InvalidCard {
        try {
            thisCard.read () ; // may throw InvalidCard exception
            pin = KeyPad.readpin () ; attempts = 1 ;
            while ( !thisCard.pin.equals (pin) & attempts < 4 )
                {   pin = KeyPad.readpin () ; attempts = attempts+1;
                    }
                if (!thisCard.pin.equals (pin))
                    throw new InvalidCard (" bad pin ") ;
            thisBalance = thisCard.getbalance () ;
            do { Screen.prompt (" please select a service ") ;
                service = Screen.touchKey () ;
                switch ( service ) {
                    case Service.withdrawlWithReceipt:
                        receiptRequired = true ;
                    case Service.withdrawlNoReceipt:
                        amount = KeyPad.readAmount () ;
                        if (amount > thisBalance)
                            {   screen.printmsg("Blance insufficirnt") ;
                                Break ;
                            }
                        Dispenser.deliver (amount) ;
                        newbalance = thisBalance - amount ;
                        if (receiptRequired)
                            Receipt.print(amount,newBalance) ;
                        Break ;
                    // other service descriptions here
                    default: break ;
                }
            } While (service != Service.quit) ;
            thisCard.returnToUser ("invalid card or PIN");
        }
        catch (InvalidCard e)
        {   Screen.printmsg("Invalid card or PIN") ;
        }
        // other exception handling here
    } //main()
} //ATM

```

---

如果读者对使用 PDL 熟悉的话,这样叙述需求能更进一步减少二义性而且更容易理解。如果 PDL 是基于实现语言的,从需求到设计就有了一个自然的过渡。PDL 使误解的可能性大大减少,做需求描述的人无须学习其他描述语言。

这种需求描述方法的缺点是:

- 这种语言表达系统功能的能力不够充分。
- 使用的符号只有那些有程序语言知识的人们才可以理解。
- 需求被看成了一个设计描述的设计,而不是帮助用户了解系统的一个模型。

该方法的一个有效使用方式是将其与结构化自然语言结合使用:一个基于格式的方法用来定义系统的总体框架,然后用 PDL 来更详细地定义控制序列和接口。

### (3) 接口系统分析

绝大多数的软件系统是要与其他已经实现的或在环境中运行着的系统进行交互的。如果新系统要同已存在的系统一起工作,已存在的系统接口必须被精确定义。这些描述在过程的早期阶段就应该给出,也可以以附录的形式在需求文档中给出。

有三种类型的接口必须定义。

- 程序接口:已存在的子系统提供的子程序接口,通过调用这些接口工程来执行子系统提供的服务。
- 数据结构:交换一个子系统到其他子系统之间的所用数据结构。基于 Java 的 PDL 可以用来描述这样的数据结构,用类来定义数据结构,用属性表示结构中的域。然而,实体-关系图表对描述数据结构更合适
- 数据的表示:一个已存在的子系统建立的数据表示。Java 不支持这么详细的表达描述,所以基于 Java 的 PDL 不适合用于此处。

形式化符号方法允许接口无二义性地定义,但是专业化的特点很难被没有受过专门训练的人掌握。虽然它是理想的工具,但很少用于实际的接口描述。作为形式化程度较低的工具,PDL 接口描述是在可理解性和精确之间的一个折中工具,但通常比自然语言接口描述更精确。

表 5-9 所示是第一个接口类型定义的例子。在这里,接口是一个打印服务器提供的程序接口,它管理请求队列将文件发送到不同的打印机上执行打印服务。用户可以检查一个打印机上的请求队列并可以将他们的打印任务从队列中撤出,他们也可以将打印任务由一台打印机转到另外一台打印机上。

表 5-9 描述的是打印服务器的一个抽象模型,并没有显示接口的细节。接口操作的功能可以用结构化自然语言给出,或使用基于 Java 的 PDL 描述。

表 5-9 打印服务器接口的 Java PDL 描述

```
Interface PrintServer {  
  // defines an abstract printer server  
  // requires: interface Printer, interface PrintDoc  
  // provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter  
  
  void initialize ( Printer p );  
  void print ( Printer p, PrintDoc d );  
  void displayPrintQueue ( Printer p );  
  void cancelPrintJob ( Printer p, PrintDoc d );  
  void switchPrinter ( Printer p1, Printer p2, PrintDoc d );  
} //PrintServer
```

## 5.2.4 系统规格说明书的编写方法

系统规格说明书即软件的需求文档，是对系统进行用户需求分析以及系统需求分析之后编写的文档，是对系统开发者要求的正式描述。它应该包括系统的用户需求和一个详细的系统需求描述。在某些情况下，用户需求和系统需求被集中在一起描述；在其他的情况下，用户需求在系统需求的引言部分给出。如果有很多的需求，详细的系统需求可能被分开到不同的文档中单独描述。

在写用户需求的时候，为了尽量减少理解偏差，这里给出书写需求应该遵守的一些简单原则：

- 设计一个标准的格式，保证所有的需求定义都按照该格式书写。标准化格式会使遗漏不易发生，需求更易检查。这种格式包括对初始需求描述的扩展、用户需求的基本原理以及详细的系统需求描述的索引。
- 使用一致的语言。尤其要区别强制性和希望性的需求。定义强制性需求时要使用“必须”，定义希望性需求时使用“应该”。
- 对文本加亮（使用黑体或斜体）来突出显示关键的需求。
- 尽量避免使用计算机专业术语。在用户需求中肯定包括对应用领域的一些描述，而这些描述包含一些详细的技术条款，这就不可避免地会用到专业术语。

需求文档有一个较广的读者范围，包括那些订购系统的高级机构管理者到负责开发系统的软件工程师。图 5-3 所示说明了文档的可能用户和他们如何使用文档。

Heninger（1980）对软件需求文档提出了六点要求：

- 应该只叙述系统外部行为。
- 应该定义实现上的约束。

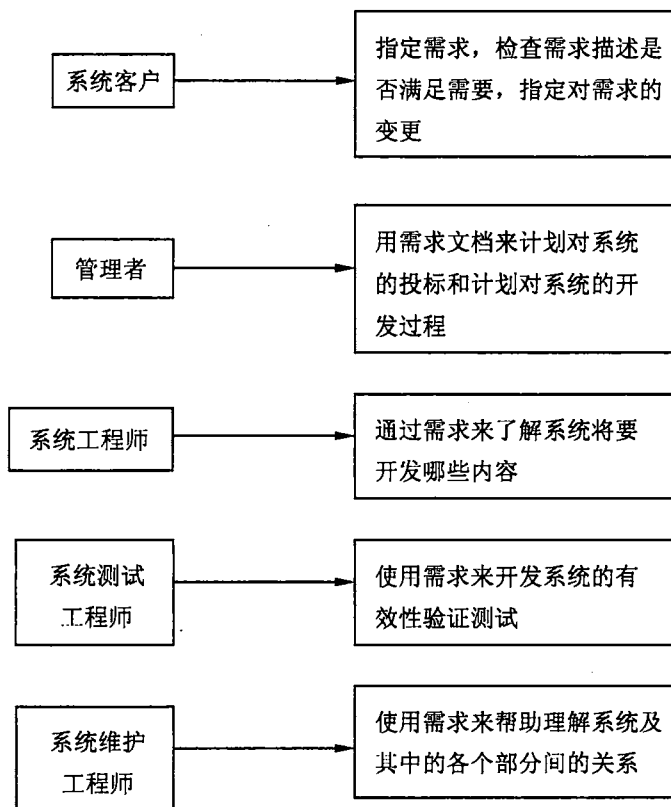


图 5-3 系统规格说明书的用户

- 应该是容易改变的。
- 应该成为系统维护人员的参考工具。
- 应该记录系统的整个生命周期。
- 应该对未料到的事件给出可接受的反应。

虽然这个陈述是在 20 年前给出的，但今天看来仍然是一个好的忠告。有时候描述系统将要做什么（即系统的外部行为）是相当困难的。由于系统设计受到现存系统的约束，所以不可避免的，系统设计是受约束的，这一点必须在需求文档中反映出来。其他的忠告，例如需要对系统整个生命周期进行记录，这一点已被广泛地接受，但是在编写需求文档时并没有被广泛地执行。

许多不同的大型机构（例如美国国防部和 IEEE）已经为需求文档定义了标准。Davis（1993）讨论了这些标准并比较了它们的内容。最知名的标准是 IEEE/ANSI 830-1993 标准

(IEEE, 1993)。Thayer 和 Dorfman (1993) 在他们最好的有关需求工程的论文集中有对该标准的一个详细描述。这个 IEEE 标准为需求文档提出了以下结构：

## 1. 引言

### 1.1 需求文档的目的

### 1.2 产品范围

### 1.3 定义、首字母缩写词与缩略语

### 1.4 参考文献

### 1.5 文档的其余部分概览

## 2. 一般描述

### 2.1 产品透视

### 2.2 产品功能

### 2.3 用户特征

### 2.4 一般约束

### 2.5 假设和依赖性

3. 专门需求 覆盖功能、非功能和接口需求。显然，这是文档中最实质性的部分，但是因为在组织的实践中存在极大的变数，对这一节定义标准的结构是不适当的。需求可能记录外部接口、描述系统功能和特性、定义逻辑的数据库需求、设计约束、系统总体特性和质量特性。

## 4. 附录

## 5. 索引

虽然 IEEE 标准不理想，但是它在如何书写需求和如何避免问题上包含很多好的忠告。它过于一般化，不足以直接拿来作为机构内部标准。然而，它可以被选择和改编来定义适合特别机构的标准。表 5-10 说明了一个基于 IEEE 标准的需求文档的结构。这里是对 IEEE 标准的扩展，包含了 Heninger 提出的系统进化的有关内容。

当然，一个需求文档中的内容是同被开发软件的类型以及开发中使用的方法紧密相关的。如果一个进化式方法用在一种软件产品开发上，需求文档将会遗漏上面提到的许多有关细节。在这种情况下，设计者和编程人员将根据他们的判断来决定如何设计系统以满足用户的需求。

相反，当软件系统是大型系统工程项目的一部分时，大系统本身包含交互式硬件和软件系统，一般就必须在细节层次上定义需求。这意味着需求文档会非常长，而且可能包含表 5-10 中的大部分章节。对于长文档，尤其需要一个详细的目录和文档索引，以便读者能快速找到所需信息。



表 5-10 系统规格说明书的结构

章 节	描 述
绪言	定义文档的读者对象, 说明版本的修正历史, 包括对新版本为什么要创建以及每个版本间的变更内容的概要
引言	应该描述为什么需要该系统。应该简要描述系统的功能, 解释系统是如何与其他系统间协同工作的。要描述该系统在机构总体业务目标和战略目标中的位置和作用
术语	定义文档中的技术术语。假设文档读者是不具有专业知识和经验的人
用户需求定义	这一部分要描述系统应该提供的服务以及非功能系统需求, 该描述可以使用自然语言、图表或者其他各种客户能理解的符号。产品和过程必须遵循的标准也要在此定义
系统体系结构	这一部分要对待建系统给出体系结构框架, 该体系结构要给出功能在各个模块中的分布。在能被复用的结构中, 组件要以醒目的方式示意出来
系统需求描述	这一部分要对功能和非功能需求进行详细描述。如有必要, 对非功能需求要进一步描述, 例如, 定义与其他系统间的接口
系统模型	这一部分要提出一个或多个系统模型, 以表达系统组件、系统以及系统环境之间的关系。这些模型可以是对象模型、数据流模型和语义数据模型
系统进化	这一部分要描述系统基于的基本设想以及预计硬件进化和用户需求改变时所要做的改变
附录	这一部分要提供与开发的应用有关的详细、专门的信息。该附录的例子是硬件和数据库的描述, 硬件需求定义了系统最小和最优配置, 数据库需求定义了系统所用的数据的逻辑结构和数据之间的关系
索引	可以包括文档的几个索引。除了标准的字母顺序索引外, 还可以有图表索引、功能索引等

在系统规格说明书的编写工程中, 需要尤其注意以下几点重点内容:

- 一个系统规格说明书描述了系统应该做什么以及定义系统运行时和实现时的约束。
- 功能需求是有关系统一定要提供的服务或者是必须执行的计算的描述。领域需求和功能需求起源于其应用领域的特性。
- 非功能需求包括对所开发系统约束的产品需求、用于开发过程的过程需求和外部需求。这些需求与系统的总体特性相关, 且作用于系统整体。
- 用户需求是为面向购买和使用系统的用户而写的。应该使用自然语言和图表的形式以使用户容易理解。
- 系统需求要达到能以准确的方式沟通系统必须提供的功能。为减少二义性, 可以采用某种结构化语言书写, 结构化语言是对自然语言给出结构化的格式约定; 或者用一种高级程序设计语言或一种特别的需求描述语言来书写。

- 软件需求文档是经过认可的系统需求描述。它应该被组织成两部分，以便既可以为系统客户使用又可以为软件开发者使用。

## 5.3 系统设计知识

### 5.3.1 传统的系统设计方法

传统的软件生命周期模型是一种顺序模型，采用自顶向下的方式把软件开发过程分为系统定义、需求分析、设计、编码、测试和维护等阶段。在开发过程中。这些阶段顺序进行，就像是一个飞流直下的瀑布，所以叫做瀑布模型。

它主要分为以下几个部分。

- 项目计划：在这一阶段对开发的项目进行可行性论证。从两个方面入手：一方面分析技术可行性，对现有软件和成熟的技术进行研究，看能否实现项目要求；另一方面分析经费可行性，看目前的经济条件能否适应项目的要求，最后写出一份任务分析书。
- 需求分析：在这一阶段要解决“做什么”的问题，要根据提出的问题写出需求分析文档，确定项目目标、系统功能、系统性能及数据在系统中的流向等，为以后的开发设计提供依据。
- 功能设计：在这一阶段要解决“怎么做”的问题，分为总体设计与详细设计两个阶段。总体设计根据前一步需求分析得出的需求分析文档将系统分成若干模块，确定模块的功能、系统的总体结构、各模块间的关联图；详细设计确定每个功能模块的名称、功能描述及算法、功能模块的输入/输出信息、功能模块之间的输入和输出接口。
- 编码：根据上一阶段得出的各个功能模块的算法，采用某种语言写出算法程序清单。这是一个将算法转换为程序的机械过程。
- 测试：写出测试计划书，描述测试规格，写出测试报告。测试可分为单元测试、综合测试、系统测试、适应测试等各种形式，从不同角度来处理问题。单元测试将上一阶段得到的程序清单按功能进行测试，使得每个功能模块对应的程序均能实现功能模块的要求；综合测试是将整个程序整体进行测试，测试程序能否实现项目要求；测试其综合性能；系统测试是从用户的角度进行的测试；适应性测试是为拓宽系统的应用而进行的测试，测试系统的可扩充性；测试阶段若不发现问题，则投入下一步处理。
- 运行与维护：运行程序是设计程序的最终目的，到此设计过程已基本完成，但由

于前面几步中可能因为考虑不周而导致在运行过程中出现各种可能的问题，这时设计者有责任进行维护，返回到某步进行处理。如此经过反复处理，可使得设计的程序能最好的满足用户的需求。

总体设计通常由以下阶段组成。

- 设想供选择的方案：设想数据流图中的处理分组的各种可能的方法，抛弃在技术上行不通的分组方法。
- 选取合理的方案：从前一步得到的方案中选取若干个合理的方案，通常至少选取低成本、中等成本和高成本的三种方案。
- 推荐最佳方案：分析员综合分析对比各种合理方案的利弊，推荐一个最佳的方案，并且为推荐的方案制定详细的实现计划。
- 功能分解：从实现角度把复杂的功能进一步分解。功能分解导致数据流图的进一步细化，可用 IPO 图来简要描述细化后的每个处理算法。
- 设计软件结构：应该把模块组织成良好的层次系统，顶层模块调用它的下层模块以实现程序的完整功能。可用层次图或结构图来描绘。
- 数据库设计：包括模式设计、字模式设计、完整性和安全性设计及优化。
- 制定测试计划：在软件开发的早期阶段考虑测试问题，能使软件设计人员在设计时注意提高软件的可测试性。
- 书写文档：应该用正式的文档记录总体设计的结果，通常应该包括系统说明、用户手册、测试计划、数据库设计结果等。

详细设计阶段的根本目标是确定应该怎样具体地实现所要求的系统，得出对目标体系的精确描述，设计出程序的“蓝图”，以后程序员将根据这个蓝图写出实际的程序代码。详细设计的工具有程序流程图、N-S 盒图、PAD 图、判定表、判定树、过程设计语言（PDL）等。

在进行详细设计的过程中，可以使用 Jackson 程序设计方法来设计。它由以下五个步骤组成：

- (1) 分析并确定输入数据和输出数据的逻辑结构，并用 Jackson 图描绘这些数据结构。
- (2) 找出输入数据结构和输出数据结构中有对应关系的数据单元。
- (3) 用特定的规则从描绘数据结构的 Jackson 图导出描绘程序结构的 Jackson 图。
- (4) 列出所有操作和条件（包括分支条件和循环结束条件），并且把它们分配到程序结构图的适当位置。
- (5) 用伪码表示程序。

也可以使用 Warnier 程序设计方法。Warnier 程序设计方法的最终目标同样是对程序处理过程的详细描述。这种设计方法由以下五个步骤组成：

- (1) 分析、确定输入数据和输出数据的逻辑结构，并用 Warnier 图描绘这些数据结构。

- (2) 主要依据输入数据结构导出程序结构, 并用 Warnier 图描绘程序的处理层次。
- (3) 画出程序流程图并自上而下依次给每个处理框编序号。
- (4) 分类写出伪码指令。
- (5) 把前一步中分类写出的指令按序号排序, 从而得到描述处理过程的伪码。

瀑布模型法有以下优点:

- 为项目提供了按阶段划分的检查点。
- 当前一阶段完成后, 只需要去关注后续阶段。
- 可在迭代法中应用瀑布法。

瀑布法有以下缺点:

- 在项目各个阶段之间极少有反馈。
- 只有在项目生命周期的后期才能看到结果。
- 通过过多的强制完成日期和里程碑来跟踪各个项目阶段。

在传统的嵌入式系统设计中, 将硬件和软件分为两个独立的部分。在整个设计过程中, 通常采用“硬件优先的原则”, 即在粗略估计软件任务需求的情况下, 首先进行硬件设计, 然后在此硬件设计平台上进行软件设计。由于在硬件设计过程中缺乏对软件架构和实现机制的清晰了解, 硬件设计工作带有一定的盲目性。它的系统优化由于设计空间的限制, 只能改善硬件与软件各自的性能, 不可能对系统做出较好的综合优化, 得到的最终设计结果很难充分利用软硬件资源, 难以适应现代复杂的、大规模的系统设计任务。

传统的先硬件后软件嵌入式系统的系统设计模式需要反复修改、反复试验, 整个设计过程在很大程度上依赖于设计者的经验, 设计周期长、开发成本高, 在反复修改过程中, 常常会在某些方面背离原始设计的要求。软硬件协同设计是为解决上述问题而提出的一种全新的系统设计思想。它依据系统目标要求, 通过综合分析系统软硬件功能及现有资源, 最大限度地挖掘系统软硬件之间的并发性, 协同设计软硬件体系结构, 以便系统能工作在最佳工作状态。这种设计方法, 可以充分利用现有的软硬件资源, 缩短系统开发周期、降低开发成本、提高系统性能, 避免由于独立设计软硬件体系结构而带来的弊端。

### 5.3.2 实时系统分析与设计

实时系统分析阶段的主要任务是确定需要解决的问题或需要完成的目标及其约束, 同时对实时系统的软硬件做全面的分析, 并对软硬件做合理的分解, 为实时软件设计打下基础。实时系统的分析需要建模 (modeling) 和仿真 (simulation), 以便系统分析人员估计“时间和大小”。目前主要有两类方法: 一类是采用数学方法, 为实时系统建模和估计时间与大小, 如 Thomas McCabe 提出的数学方法, 该方法使分析者能够为实时系统的硬件和软件成分建模, 用概率的方法表示控制, 采用网络分析、队列和图论及 Markov 数学模型得

出系统时序和资源大小；另一类是采用建模和仿真工具，建立各种形式化或非形式化模型，不仅可以分析系统的性能，还能建立一个可执行的原型，从而了解系统的行为。

实时系统设计阶段的主要任务是如何在给定的约束条件下完成系统的目标，重点在于实时软件的设计。实时软件的一些设计方法是非实时设计方法的扩展，如扩展数据流、数据结构和面向对象设计方法；另外一些则是完全不同的方法，如运用有限状态机、Petri 网、消息传递系统或以某种专门语言为基础的方法。

目前工业界存在着许多种实时系统开发的方法学，即一些格式化的、逐步细化的指南，引导用户走过分析、设计、实现和测试各阶段，给用户提供一个整套实时软件开发的指导。它们大致可以分为结构化的方法、面向对象的方法、基于组件的方法及其他方法四种。由于实时系统的分析与设计二者是紧密联系的，因此，将实时系统的分析与设计方法一并进行讨论。

### (1) 结构化的方法

在传统的结构化方法基础上，已经提出了不少针对实时系统的多任务分析设计方法，如 RTSAD (Real-Time Structured Analysis and Design)、DARTS (Design Approach for Real-Time Systems)、JSD (Jackson System Development)、CODARTS (Concurrent DARTS)、SCR (Software Cost Reduction) 等，它们本质上是结构系统分析、数据流分析、事务分析向实时软件开发的扩展。其中，RTSAD 是满足实时系统的结构化分析设计方法，不适于处理任务结构方面的内容；DARTS 弥补了 RTSAD 的不足，但在信息隐藏方面不如面向对象方法；CODARTS 集中了上述各种方法的长处，实现了信息隐藏技术，并适于处理任务结构；SCR 是一个基于信息隐藏概念的实时设计技术。

下面以 DARTS 为例，详细介绍结构化分析设计方法的开发过程。

DARTS 是 SAISD (System Analysis and System Design 结构化分析和设计) 方法在实时系统开发中的应用，按照软件生命周期进行软件开发。DARTS 通过将系统分解成任务和定义任务接口的方法来扩充结构化分析/设计的方法，使得实时应用系统具有并行处理的能力。该开发方法的关键是将一个系统分解为并行的任务，并定义任务间的接口。

DARTS 采用一种增量式开发过程，各阶段的主要任务如下。

- 需求分析和规范：分析用户需求，定义系统功能需求和 I/O 接口。
- 系统设计：利用数据流程图，将系统划分为任务（并行进程），同时定义任务接口。
- 任务设计：将每个任务划分为模块，同时定义模块接口。
- 模块构造：对每个模块进行详细设计、编码和单元测试。
- 进程和系统集成及接口测试：对每个任务的模块进行集成和测试，然后依次对系统中的任务逐步进行集成和测试。
- 系统测试：验证整个系统或主要子系统同功能规范的一致性。与需求分析和设计

过程反向进行,从过程、模块逐步向模块间的联合测试过渡,再到系统集成测试。测试方法通常是交叉测试,从主机上的仿真测试到目标机上的实际测试,测试系统是否满足实时性、可靠性、可调度性等要求。

- 确认测试:由最终用户进行该项测试,检查系统实现功能是否满足用户需求。

## (2) 面向对象的方法

实时系统开发是面向对象技术的主要应用领域之一,采用面向对象的分析设计方法,便于大型复杂的实时系统的分解和设计。目前已存在不少针对实时系统的面向对象分析设计方法,如OOD(Object-oriented Design)方法侧重于把系统划分为对象,实现信息隐藏,但在处理任务结构方面仍显不足;Octopus(Object-oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion)方法是基于早期的OMT(Object Modeling Technology)和融合方法(Fusion Method)的面向对象方法;ROOM(Real-time Object-Oriented Modeling)方法主要着重设计和实现阶段,为描述构成系统的对象(或者角色)的行为定义了一套文本和图形语言;Shlaer-MellorOOA/OOD(Object-Oriented Analysis/Object-Oriented Design)方法用于从结构化开发技术方便地过渡到面向对象的开发技术,该方法引导开发者走过分析、设计和实现阶段;GOO(Graphic Object-Oriented)方法采用一套图形标识符建立系统的面向对象模型,使用相应的图表工具可将这些图表直接转换为C代码;ObjectGeode既是方法学又是CASE工具软件,由Veri log公司开发,是目前实时系统开发中较为流行的一套方法学。

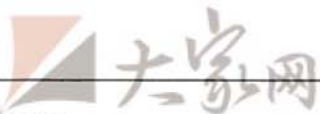
下面以ObjectGeode为例,详细介绍面向对象分析设计方法的开发过程。

ObjectGeode方法从需求分析到详细设计等各个开发阶段都使用面向对象的形式化语言来实现,不但能进行信息隐藏,适合于处理多任务结构,还使实时系统的开发过程更加规范,并能借助工具软件,实现目标代码生成过程的自动化。开发过程中所使用的形式化语言包括UML、MSC和SDL。UML是由OMG管理面向对象的形式化语言规范,用来描述客观实体,以及系统与环境或子系统之间的交互关系。UML综合了几种流行的面向对象方法学中的标识符,它把握着具正标准的面向对象的模型语言。目前在UML中已经添加了一些元素,使其适合于实时系统的开发。MSC是ITU-T 2.120标准,可用于描述实时系统与环境、系统各组件之间的通信行为。SDL是ITU-T定义的面向对象的形式化语言,适合于描述事件驱动交互式应用。

ObjectGeod方法详细说明了一种迭代式的开发过程,包括以下主要步骤。

① 需求分析。需求分析主要对用户需求和与系统交互的环境进行建模,包括对象建模和动态建模两个过程。对象建模刻画了系统的静态模型,用来描述系统的静态关系,需要确定出与系统有关的实体,并以类图和实例图的形式表示出来;动态建模通过用户实例建模来进行,以描述系统的动态行为,需要确定出系统与环境、系统内部各组成部分之间





的交互关系。对象模型用 UML 类图描述，动态模型用 MSC 图描述。

② 设计。概要设计定义了系统的整体逻辑结构（软件体系结构），用 SDL 的体系结构图和互连图来描述。详细设计用来细化体系结构中的软件对象，需要确定两类对象：并行对象和被动对象。并行对象包含在系统中能够并行运行的控制进程，被动对象则作为并行对象所使用的资源而存在。详细设计使用 SDL 和 UML 来描述，并行对象通过 SDL 进程图来描述，被动对象使用 UML 类图来描述。

③ 目标化阶段。用来生成最终应用，并使开发出的实时系统能够与目标环境下的实时操作系统协调运行。ObjectGeode 工具软件具有直接根据分析和设计阶段建立的模型生成 C 或 C++ 代码的功能，同时实现同 RTOS（包括 Chorus、p50S、VRTX、VxWorks、Win32 以及几种 UNIX 变种等）的绑定，将 SDL 信号转化为 RTOS 消息、进程转化为 RTOS 任务。

④ 测试。在 SDL 和 MSC 分析设计中自动生成测试用例，并转化为具体用例，对系统进行验证测试。该测试完成后，交由系统用户进行确认测试，该测试不考虑系统体系结构，侧重于实际应用提供的功能，以及系统的质量和性能。确认测试主要依靠需求分析阶段建立的用户实例模型。

### （3）基于组件的方法

在嵌入式系统中，组件的表现形式多种多样，如 RTOS 内核（或执行体 Executive）、TCP/IP 协议栈、嵌入式 Web Server、嵌入式数据库、嵌入式 CORBA、软件 Modem、MPEG 算法等都是面向嵌入式系统的组件。随着嵌入式应用程序对分布式计算需求的不断增强，嵌入式 CORBA 组件、JavaBeans 组件以及 Windows CE 平台上的 COM+ 组件也将得到进一步的应用。

组件以其清晰的功能和良好的接口，日益成为提高嵌入式软件开发效率和改善嵌入式软件开发质量的一种重要手段。用户可将其直接集成到自己的嵌入式应用中，而不需要从头开发。关于组件的使用和创建也有一套方法和过程，包括如何定义一个组件，如何为组件创建一个好的 APT，如何选择、集成和测试第三方组件等。

平台开发模式（即基于平台观念的系统分析设计方法）是在建立各种平台（厂家、产品或应用软件平台，由应用框架、组件、IP 核、开发工具集等组成）基础上，通过集成已存在的组件，运行系统开发和产品设计的方法。这种基于组件的方法学既非 Top-Down 也非 Bottom-Up。前者以应用的规格指标为出发点，逐渐推进设计，求得微结构成本最低的解决方案，后者则首先决定微结构。而基于平台的设计是从处于顶层的应用与底层的微结构之间的平台开始设计，在这个意义上它是一种新的设计方法。

### 5.3.3 软硬件协同设计方法

当协同设计一个时间紧急的复杂系统时，可以将软件和硬件两个生命周期同时进行。

在进行软硬件协同设计时，完成设计的软件人员必须了解硬件设计的种类和硬件设计小组可能遇到的问题种类，还必须了解硬件的可行性及其性能。

软硬件协同设计在实际应用中表现为软硬件协同设计平台的开发。经过评估后的设计可以进行软硬件划分，产生硬件描述、软件描述和软硬件界面描述三个部分，以及各个部分的具体实现并优化。最后进行硬件综合、软硬件集成和系统仿真测试。

软硬件协同设计流程从目标系统构思开始。对一个给定的目标系统，经过构思，完成其系统整体描述，然后交给软硬件协同设计的开发集成环境，由计算机自动完成剩余的全部工作。一般而言，还要经过模块的行为描述、对模块的有效性检查、软硬件划分、硬件综合、软件编译、软硬件集成、软硬件协同仿真与验证等各个阶段。软硬件协同设计流程如图 5-4 所示，其中软硬件划分后产生硬件部分、软件部分和软硬件接口界面三个部分。硬件部分遵循硬件描述、硬件综合与配置、生成硬件组件和配置模块；软件部分遵循软件描述、软件生成和参数化的步骤，生成软件模块；最后把生成的软硬件模块和软硬件界面集成，并进行软硬件协同仿真，以进行系统评估和设计验证。

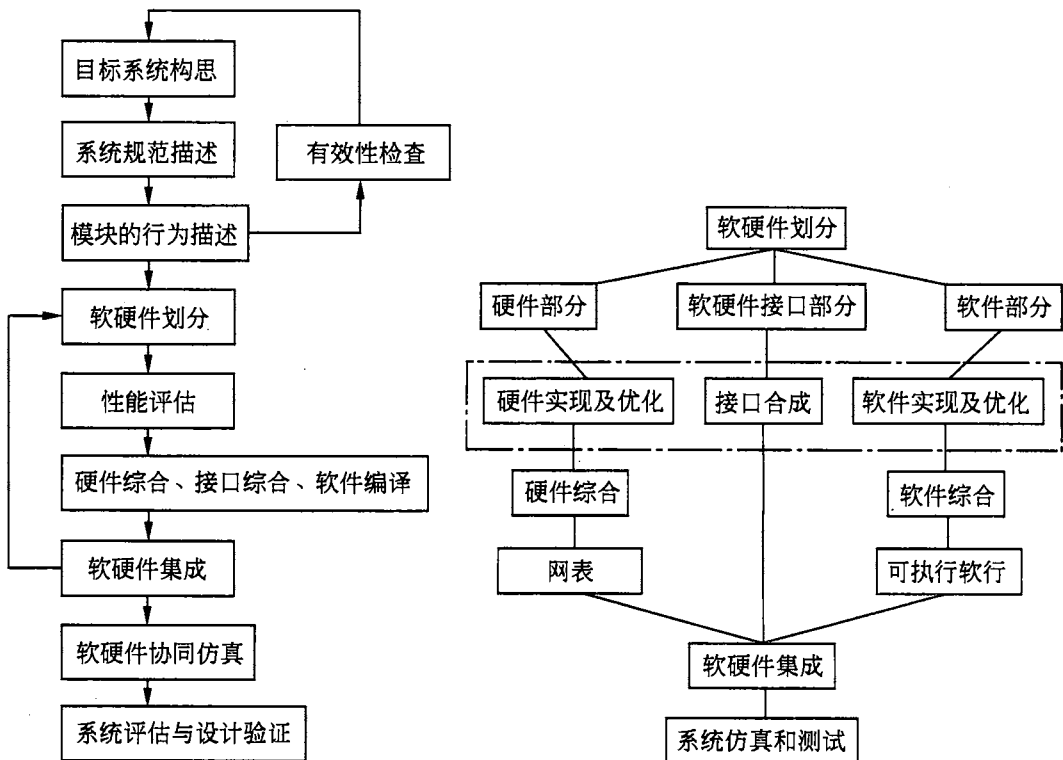


图 5-4 软硬件系统设计流程



软硬件协同设计最主要的一个优点就是在设计过程中，硬件和软件设计是相互作用的，这种相互作用发生在设计过程的各个阶段和各个层次。设计过程充分体现了软硬件的协同性。在软硬件功能分配时就考虑到了现有的软硬件资源，在软硬件功能的设计和仿真评价过程中，软件和硬件是互相支持的。这就使得软硬件功能模块能够在设计开发的早期互相结合，从而及早发现问题及早解决，避免了在设计开发后期反复修改系统以及由此带来的一系列问题，而且这样有利于挖掘系统潜能、缩小产品的体积、降低系统成本、提高系统整体性能。

总的来说系统设计可以分为以下几个部分。

#### (1) 系统描述

系统开发过程的目标是，生成一个完全通过测试和验证的系统。

所谓系统描述就是将系统功能全面表述出来，是完整的规范和系统需求，如产品功能和任务、交付时间表、产品生命周期、系统负载、人机交互、操作环境、传感器、功耗需求和环境、系统开销等。在这一阶段中系统建模是个重要环节。系统建模就是建立系统的软硬件模型，优化系统描述的过程，对于不同的应用建立模型的方式和使用的描述语言也可不同。系统建模对系统的性能影响极大，它不仅直接影响系统的造价、开发、周期、工作、环境等外观因素，而且与系统的稳定性、可维护性等隐性指标密切相关。系统建模可由设计者手工完成，也可借助 CAD 工具实现。传统的系统建模方法基本上是在系统的输入和输出之间建立一种简单的对应关系，并且常常使用非正规语言甚至是自然语言来描述。这就容易导致系统描述不准确，在后续过程中必须反复修改系统模型，从而产生使系统设计复杂化等问题。软硬件协同设计就是要全面描述系统功能，精确建立系统模型，深入挖掘硬件之间的协同性，以便系统能够稳定高效的工作。

具体来说系统模型应明确体现下列因素。

- 性能描述：性能描述反映的是系统的整体面貌及体系结构。它应该明确地或隐含地说明系统 I/O 及相关的中间状态，以及其相互之间的关系。性能描述应系统、全面，尽可能地把所有可能的因素都表述出来。
- 功能特点：这是系统应该完成的基本功能，它应明确表述各项功能特点与系统 I/O 及中间状态之间的关系，以便对系统描述进行核实。
- 技术指标：它们是评价系统质量的指标体系，常常表现为价格、速度、字长、可靠性等具体项目。
- 约束条件：它将明确规定技术指标的适用范围、系统的工作环境要求及系统性能的缺陷和不足等。它们是确保系统正常工作的环境要求，是系统性能好坏的具体体现。在系统建模时要选择合适的描述语言和选用合理的处理方法，传统的描述方法已经不能满足软硬件协同设计的要求，一些新的技术已经在研究开发中。

## (2) 系统设计

在开发软件时,我们需要知道所需的软件模型是什么、开发过程的软硬件说明是什么。这可以参考各种不同的系统硬件单元和硬件要素描述。

系统设计是指一个把抽象的系统描述,转换成具体的可操作的系统的过程。构建步骤的过程对于嵌入式系统来说,系统设计可以分为软硬件功能分配和系统映射两个阶段。

- 软硬件功能分配:就是确定哪些协调功能由硬件模块来实现,哪些系统功能由软件模块来实现,即确定系统功能的具体实现形式。由于硬件模块的可编程性和嵌入式系统的变异性,软硬件的界限已经不十分清楚,所以软硬件的功能划分是一个复杂而艰苦的过程。一方面是由于软硬件划分的研究工作还处在初级阶段,另一方面则是由于这一问题内在的复杂性在进行软硬件功能分配时,既要考虑市场可以提供的资源状况,又要考虑系统造价、开发周期等因素。
- 系统映射:就是根据系统描述和功能分配,选择确定系统的体系结构,它将决定产品最终的整体面貌。具体地说,这一过程就是确定系统将采用哪些硬件模块,如微处理器、微控制器、存储器、ASIC、DSP、FPGA、I/O 接口部件等软件模块;驱动程序等功能程序以及软硬件模块之间的联系媒体,如总线、共享存储器数据通道等。

## (3) 基于 UML 来开发

随着嵌入式系统的日趋复杂化,较多的系统都需要由一个团队共同完成,因此,团队成员之间的相互合作,软硬件之间的协同开发,乃至开发人员和客户之间的交流都需要有一个统一的标准作为基础。UML 正是这样一种标准的系统建模语言,它详细描述系统的内容和工作方法,先进行系统建模后再编写代码,在开始阶段就保证了系统结构的合理性。UML 系统模型包含许多不同框图,使项目小组可以从不同角度了解整个系统。另外,UML 可以用统一的形式表现软件和硬件,支持循环迭代并可多次修改软硬件方案直到满足要求,可实现软硬件协同设计。特别要指出,UML 是一种语言,不是方法,它独立于开发过程。

UML (Unified Modeling Language) 是一种定义良好、易于表达、功能强大且普遍适用的面向对象和基于构件的系统建模语言。它扩展了现有方法的应用范围,不仅可建立软件系统的模型,还可建立非软件系统的模型,可广泛用于描述系统软件、嵌入式系统、企业机构或业务过程等。在嵌入式系统的协同设计中引入 UML “类图”、“用例图”、“对象图”、“顺序图”、“状态图”有助于原理设计的开发,以及获取应用软件和硬件的结布局。这些图为系统的开发提供了多种图形表达形式,应用于建模的不同阶段。

基于 UML 的嵌入式系统开发方法支持需求、分析、设计、实现、测试的循环迭代,使用面向对象思想,通过细化分析和设计阶段的步骤,使得整个过程更有条理、充实,更适合于多任务的嵌入式系统开发方法的需求、分析、设计过程。被细化后分别包括了以下

几个步骤:

① 需求阶段明确了系统所要实现的功能以及所要达到的性能,是整个系统开发的目标。它分为以下两个阶段。

- 功能性需求:功能性需求是系统功能的陈述,明确系统应该提供什么功能。在UML中是用用例图来描述系统功能的。这个阶段的各个用例只是对系统功能的大致划分,主要目的是为后面的系统分析做基础。
- 非功能性需求:非功能性需求是系统的特定特性。非功能性需求为确定系统的结构和系统选用的技术等进行了约束。

② 分析阶段主要是精化和结构化需求,清楚地描述了系统内部,是设计阶段的基础。在系统分析阶段,通过细化和结构化系统需求,可将系统需求转换成系统中的结构、类、对象和关系等实体元素,并从静态和动态两个角度来清楚描述这些实体元素。它分为两个步骤:

- 系统架构分析:运用面向对象技术描述系统的静态结构。系统结构分析是对系统元素静态的描述,它在系统需求的基础上确定系统的总体架构及内部对象。它用部署图来描述系统的物理架构,然后用类图来描述系统静态的对象结构及其相互关系。从用例图中我们可分解出一些类,并将这些类之间的结构描述出来。
- 系统行为分析:从动态的角度描述系统的对象间相互作用的特性。系统行为分析就是从多个角度来描述所研究系统的动态部分。我们可用状态图描述系统的状态行为,然后根据系统内部所具有的行为来定义和精化类的操作,另外也可用顺序图和协作图从不同的角度来显示动态的信息流。根据嵌入式系统的特点,在此处,状态图不但包括嵌套层次结构状态的概念,还可用并发的概念来表示那些可以和其他状态同时处于活动状态的独立状态。

③ 设计阶段是在对系统各方面有了解的基础上来确定特定的解决方案。它分为两个步骤:

- 分层结构设计:确定了具体实现时软件和硬件的最佳分界。系统具体实现时还是需要将软、硬件分开实现,我们也要在系统设计阶段对软硬件层次进行划分。若一次划分不能满足要求,也可以通过迭代在以后的循环中尝试多种方案,直到满足要求。在系统结构分析中用类图所做的统一描述涵盖了软件层和硬件层共同组成的系统结构,所有软件层和硬件层都是由类图中提取而来的,但类图中既可由软件实现又可由硬件提供的一部分内容,则要根据性能、价格、规格大小等因素来加以选择。用部署图描述的系统硬件层将类图中的数据处理对象,即软件层中的操作系统所具有的任务映射到了处理器的各个线程,并且还设置了每个线程的优先级。

- 详细设计：在软件方面是深入到了系统低层信息，如操作的属性、类的流程等；硬件方面则是到了设计具体电路板的阶段。详细设计是一次循环中需求、分析、设计的最后一步，指定了细节问题，明确了单个对象的范围、内部数据结构和算法的实现等。先前已对类的属性和操作做了定义，而在详细设计中，为了编写代码，必须对每个类中定义的操作的各个属性（包括它的类型和初始值等）填补完整。因为此时的类图是为软件编程准备的，所以应根据体系结构设计过程中组件图的内容重新进行整理，保留并细化由软件实现的所有类。依照这些类的行为流程图，在编程阶段就可以容易地实现代码，并且由于有了统一的设计决策，即使是由不同的编程人员编写，最后的代码体现出的思路也都是大同小异的，也方便非开发人员了解和维护系统。

利用面向对象的概念将系统分成了相互关联却又较独立的模块，一方面方便了系统开发时的迭代过程以及系统的后期维护，设计人员可以根据不同的新的需要对各个步骤中相应部分进行调整来实现改进，这样就可以大量减少重复分析或设计的过程；另一方面，对象概念可以同嵌入式系统中的任务概念很好地映射起来。任务可看成是由一个或多个对象协作而成的，在分析、设计过程中确立对象的同时也就确定了系统的多个任务，为嵌入式系统的多任务特性提供了很好的支持。

#### （4）系统评价

系统评价是检查确认系统设计的正确性的过程，即对设计结果进行正确的评估，以避免在系统实现过程中发现问题时，再进行反复修改的弊端。目前系统模拟仍是系统评价的重要手段，但由于嵌入式系统的变异性，实现对系统软硬件的协同模拟是一件具有挑战性的工作。因为在系统模拟过程中，使用主机 CPU 往往要比真正在嵌入式系统中使用的 CPU 要快得多，这就难以保证模拟结果的真实性。其次由于模拟的工作环境和实际使用时的工作环境差异很大，软硬件之间的相互作用方式及作用效果也就不同，这就难以保证系统在真实环境下工作的可靠性，所以系统模拟的有效性是有限的。

近年来形式化评价技术进步迅速，并越来越引起人们的重视，特别是对安全性要求较高的嵌入式系统有其独到之处。形式化评价技术通过建立精确的数学模型，利用数学手段检测系统的正确性，所以对系统中的不确定因素及隐性指标的检查有特殊效果。

#### （5）综合实现

软硬件综合实现，是指软硬件系统的具体制作设计结果经确认后，即可按设计要求进行。系统制作即按照前述工作的要求，定制硬件、设计软件并使它们能够协调一致地工作。在此过程中应最大限度地利用现有资源，即购买那些已有的功能模块采用 FPGA，设计那些买不到的功能模块，并将它们有机地结合在一起使其能够高效工作。

近年来随着CAD技术的不断发展，计算机辅助协同设计技术也有了长足发展。计算机

模拟综合等辅助设计技术的进步,对软硬件协同设计技术无疑是一个极大的帮助和推动。它不仅简化了系统设计的难度,提高了系统设计的效率,而且也使软硬件的协同设计逐步走上正规化的道路,所以计算机辅助协同设计工具的开发和推广应用,将大大推进软硬件协同设计技术的发展。目前已有一些试验性软件支持工具投入试用,但这方面的工作还需进一步发展和完善。

#### (6) 实现工具和测试

在开发过程的编辑-测试-调试周期中,我们可以使用一些工具来提高效率。

##### ① 使用目标系统

目标系统包含处理器、存储器、外围设备和接口,是为了获得实际的嵌入式系统而复制的系统。与最终系统不同,目标系统和计算机结合起来可以很好的工作,如同一个独立的系统。在开发阶段可能需要重复将代码下载到其中。目标系统或者它的副本只在后面作为嵌入式系统工作。

为目标系统编写的代码必须嵌入到存储器、内存、EPROM 或者 EEPROM 中,经模拟器和调试工具反复编写和修改,直到它可以按照规范中的要求进行工作,才进行嵌入操作。此后,设计者可以简单地将其复制到最终系统或者产品中。最终的系统可以使用 ROM 代替目标系统中的内存、EPROM 或 EEPROM。

##### ② 仿真器和 ICE

我们也可以不使用目标系统,而采用一个保持与特定目标系统和处理器无关的独立单元,即使用仿真器或者 ICE。它为在单一系统上开发不同的应用程序提供了很大的灵活性和便利性。

仿真器使用由微控制器和处理器自身组成的电路。仿真器可以模拟具有扩展存储器的目标系统。ICE 使用另外的带卡的电路,这个卡通过插座同目标处理器(或电路)相连。ICE 或者仿真器在开发阶段结束后就不再使用。通过复制使用 ICE 开发的代码构成电路。

##### ③ 逻辑探测器

逻辑探测器是一种简单的硬件测试设备,它是一个类似 LED 设备的手持笔,可以用于研究端口上的长延时效应。

##### ④ 示波器

示波器是带有显示屏的一种显示装置,它显示两个信号的电压随时间变化的情况。它还可以显示模拟信号和数字信号随时间变化的情况。运行的时钟会在示波器上显示其状态,在相连的两个上升沿之间的水平间隔给出时钟的周期。示波器还可用作噪声检测工具和电压表,还可检测一个时钟周期内 0 和 1 两个状态之间的突变。

##### ⑤ 逻辑分析仪

逻辑分析仪是一个比示波器功能更强大的工具,它可以检查载有地址数据、控制位和

时钟的多个输入线路。它可以同步连续的收集、存储和跟踪多路信号，可以调试实时触发条件。

#### ⑥ 位率测量仪

位率测量仪是一种测量设备，可以在预先选择的时间区间查找 1 和 0 的数量，可以测量网络的吞吐量。

#### (7) 嵌入式软件开发过程中的设计问题

嵌入式系统是具有额外的程序设计需求，所以设计时要考虑一些额外的问题。

- 在需求和规范的分析中存在的问题。出现实时事件时，程序流控制规范、每个软件功能可接受的最大响应时间、事件的最大延时、中断服务例程可接受的最大延迟、软件组件和中断优先级。如何向那些具有更短时限的中断分配优先级？可靠性与可接受的容错性如何？
- 设计和实现中的问题。处理器、存储器和硬件的选择；单处理器和多处理器的选择；系统中存储器和功耗的优化设计，决定是否使用高速缓存及如何使用；是让软件开发小组设计 RTOS 还是使用已经测试和调试好了的 RTOS；测试方法、调试断点以及宏的选择。
- 系统集成中的问题。使用适当接口的统一系统的集成和开发；先对接口进行验证，然后对集成系统进行验证。
- 测试中的问题。由于嵌入式实时系统输入、事件以及中断是与时间相关且异步的，所以嵌入式实时较难测试。需要白盒测试、黑盒测试以及烟雾测试等。需要处理的问题有：测试策略的选择、硬件和软件集成的系统测试、测试适当的中断优先级分配、对任务和 ISR 的吞吐量的测试、IPC 的测试以及软件在硬件中的可移植性的测试等。
- 选择合适的平台。平台由许多单元组成：处理器、ASSP、片上系统、存储器、总线、RTOS、软件语言、代码生成工具和系统的其他硬件单元。设计者选择合适的单元以最终得到合适的平台和开发工具。
- 处理器的选择。一般使用带有微处理器、微控制器或者 DSP 的系统。另外，PLC 操作速度很慢，计算能力也很差，但是有非常强大的多 I/O 接口，具有系统相关的可编程能力，有些时候可以代替处理器。
- 需要考虑必需的特性和因素。这包括使用 8 位、16 位还是 32 位的芯片，以及存储器的大小、同步通信是全双工还是半双工、输入捕捉和输出比较、功耗大小等。
- 软硬件权衡。在硬件和软件之间需要权衡。硬件中的某些子系统、实时时钟、脉宽调制器、定时器和串行通信也可以通过软件实现。硬件实现不仅可以加快操作速度，也会增加开销；软件实现易于随新需求更改，而且具有复杂的编程能力等。

- 系统性能指标。系统性能指标可以定义为：在使用最少量的存储资源、功耗和设备，最少的设计工作量，以及每个资源优化使用的情况下，满足所需功能和规范的能力。最好的嵌入式软硬件是那些可以在不同的性能度量之间获得平衡的软硬件。

#### (8) 几种协同设计方法

##### ① VULCAN

VULCAN 是 Stanford 大学开发的软硬件协同设计工具。系统基于高层综合工具 olympus 开发，采用类 C 语法的硬件描述语言 HardwareC 作为系统描述手段，并将 Hard2wareC 描述编译为控制/数据流图作为综合工具的内部表示。其设计目标是带有单一 DLX 处理器、单层次存储器、多个硬件单元、单总线的嵌入式系统。但实现软硬件协同综合方法较简单，它采用贪婪算法将可变更为软件实现的部分从硬件描述中划分出来。VULCAN 可以看做是考虑了软件因素的高层综合工具。

##### ② COSYMA

COSYMA 是由德国 Braunmschweig 大学设计的，主要面向软件。它的目的是计算优化的软硬件划分以通过协处理器获得最大的加速比。在 COSYMA 中，它允许定义并发和时间约束。软硬件划分由基于使用估计调度时间的模拟退火算法自动实现。COSYMA 主要的缺点是处理器和协处理器不能并行工作。

##### ③ POLIS

POLIS 是由美国 Berkeley 大学开发的，其系统的核心是 CFSM (Codesign Finite State Machine)，它描述软硬件模块时没有差别，差别在于系统状态转换的延迟上。

POLIS 采用 Esterel 作为系统输入语言，并采用扩展的有限状态自动机模型 CFSM 进行系统描述。POLIS 主要研究高级语言的转换、规则检查、软硬件界面综合等技术，软硬件划分算法和调度留给设计者手工进行。由于采用了基于语义明确的形式化系统模型，POLIS 实现了划分后的软件、硬件以及软硬件接口的自动综合。POLIS 也采用 Ptolemy 进行协同仿真。由于采用有限状态自动机作为系统模型，POLIS 适合设计实时控制领域的嵌入式系统。

##### ④ VCC 和 N2C

Cadence 的 VCC 工具和欧洲 CoWare 公司的 N2C 工具中包含了交互式软硬件划分功能。CoWare N2C 工具强调支持开发人员进行系统设计空间探索，使用 N2C 工具，开发者将硬件-软件模块用 C/C++ 语言设计为行为级描述，然后通过几个层次——无时序模型、指令精确模型和总线周期精确模型的仿真来探究不同划分方案。仿真过程应用了指令集模拟器和硬件描述语言模拟器，最大的优点是支持在设计中集成 IP 核。

## 5.4 系统实施知识

### 5.4.1 系统架构设计

#### 1. 系统架构设计在软件生命周期中的作用

在设计阶段，我们将构造系统，并实现所有需求（包括非功能性需求和其他约束）的系统组织（包括系统构架）。

设计工作主要集中在细化阶段的末期到构造阶段的初期（如图 5-5 所示），在这一过程中将产生合理而稳定的构架，并应首先创建模型蓝图。在随后的构造阶段中，当系统已经稳定并且已经很好地理解需求以后，工作的重点将转向系统的实现。

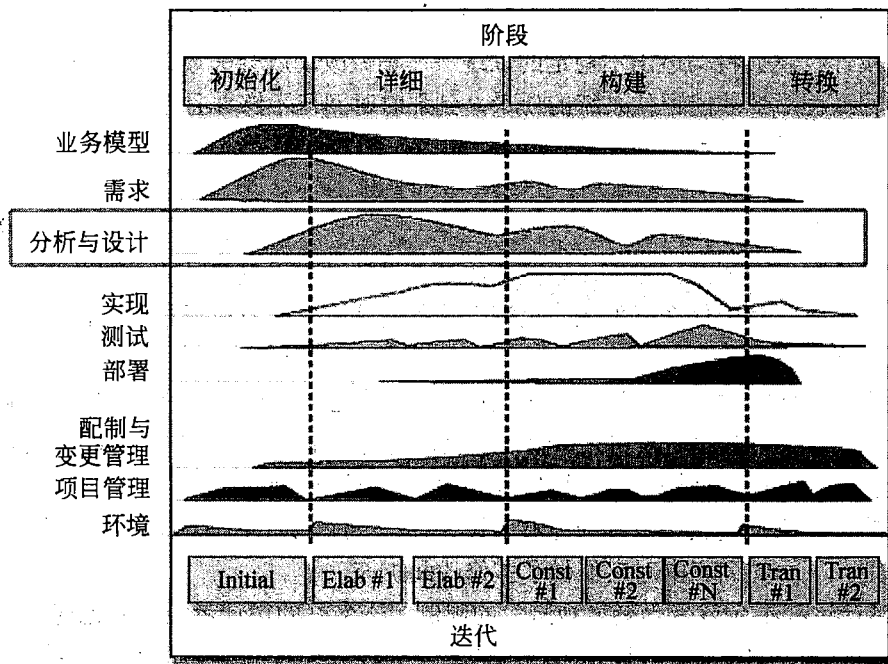


图 5-5 系统架构设计

系统架构设计的目的在于：

- 深入理解与非功能性需求和约束相联系的编程语言、构件重用、操作系统、分布与并发技术、数据库技术、用户界面技术等相关问题。
- 通过对单个子系统、接口和类的需求捕获，为后续的实现创建适当的输入和出



发点。

- 能够把工作划分成更易于管理的各个部分，并且尽可能地由不同的开发组并行的开发。
- 在软件生命周期的早期捕获子系统之间的主要接口。这一点在理解系统架构和使用接口作为保持不同开发组之间的同步手段时是很有用的。
- 通过通用的符号，可以可视化地刻画和思考设计。
- 建立对系统实现的无缝抽象，把实现看成设计的直接简化。它不改变结构，只填入“血肉”，这使得应用代码生成以及在设计和实现之间的双向工程等技术成为可能。

软件设计的重要性可以用一个词来表达——质量。设计是在软件开发中形成质量的地方，设计为我们提供了可以用于质量评估的软件表示，设计是我们能将用户需求准确地转化为完整的软件产品或系统的唯一方法。软件设计作为所有软件工程和软件维护步骤的基础，没有设计，我们将构造出不稳定的系统——稍做改动就会失败；难于测试的系统；直到软件工程过程后期才能评估系统的质量，到那时时间已不够并且已经花销了很多经费。

## 2. 系统架构设计原则和概念

### (1) 设计的原则

软件设计既是过程又是模型。设计过程是一系列迭代的步骤，它们使设计者能够描述要构造的软件的所有侧面，然而，要注意的是，设计过程不仅仅是一本清单，创造性的技能、以往的经验、对于什么能形成“良好”软件的感觉，以及对质量的全部责任是设计成功的关键因素。

设计模型和建筑师的房屋设计图是类似的，它首先表示出要构造的事物的整体（例如，房屋的三维表示），然后逐渐精化事物，以提供构造每个细节（例如，管道布置）的指南。类似地，软件的设计模型提供了计算机程序的一系列不同视图。

基本的设计原则使软件工程师能够为设计过程导航。Davis[DAV95]提出了一系列软件设计的原则。下面的列表中对它们做了些修改和扩充。

- 设计过程不应该受“隧道视野”的限制。一名好的设计者应该考虑替代的手段。
- 设计对于分析模型应该是可跟踪的。因为设计模型的单独一个元素经常会跟踪到多个需求上，所以对设计模型如何满足需求进行追踪是必要的。
- 设计不应该从头做起。系统是使用一系列设计模式构造的，很多模式很可能在以前就遇到过。这些模式通常被称为可复用设计构件。时间短暂而资源有限！设计时间应该投入到表示真正的新思想和集成那些已有模式上去。
- 软件设计的结构应该（尽可能）模拟问题域的结构。
- 设计应该表现出一致性和集成性。如果一项设计整体上看上去像是一个人完成的，

那它就是一致的。在设计工作开始之前，设计小组应该定义风格和格式的规则，如果注意定义了设计构件之间的接口，那么，设计就是集成的。

- 设计应该构造以适应修改。
- 设计应该构造以使得即使遇到异常的数据、事件或操作条件时也能够平滑、轻巧地降级。设计良好的计算机程序应该从不“彻底崩溃”，它应该设计出适应异常的条件，并且即使它必须中止处理时，也要采用优雅的方式。
- 设计不是编码，编码也不是设计。即使在为程序构件构造详细的过程设计时，设计模型的抽象级别也比源代码要高，在编码级别上做出的唯一设计决策是描述能使过程性设计比编码的小的实现细节。
- 在创建设计时就应该能够评估质量，而不是在事情完成之后。
- 应该复审设计以减少概念性（语义性）错误。有时人们在复审设计中倾向于注重细节，只见树木不见森林。在关注设计模型的语法之前，设计者应该确保已经检查过设计的主要概念性元素（忽略、含糊性、不一致性）。

正确应用上述设计原则时，软件工程师创建的设计就会展现出外部和内部的质量因素[MEY88]。外部质量因素是那些用户能轻易观察到的软件特性（例如，速度、可靠性、正确性、可用性）；内部质量因素对软件工程师是重要的，它们能导致技术角度上的高质量设计。要取得内部质量因素，设计者必须理解基本的设计概念。

## （2）设计的概念

它是在过去 30 年中发展起来的一套基本的软件设计概念。尽管对于每一种概念的兴趣程度在几十年中一直变化着，但它们都经历了时间的考验，每一种概念都为设计者提供了比应用软件更加复杂的设计方法的基础。

### ① 抽象

“抽象”的心理学观念使人能够集中于某个一般性级别上的问题，而不去考虑无关的低层细节。使用抽象还能使人能用问题环境中熟悉的概念和术语工作，而不必将它们转换成一种不熟悉的结构。

当我们考虑任何问题的模块化解决方案时，可以给出许多抽象级别。在最高的抽象级别中，解决方案使用问题环境的语言来进行概括性的术语描述，在低一些的抽象级别中，会有更加面向过程的倾向。为了描述解决方案，要一同使用面向问题的术语和面向实现的术语。最后，在最低的抽象级别中，用能够直接实现的方式描述解决方案。软件工程过程中的每一个步骤都是软件解决方案抽象级别上的求精。在系统工程时，软件划分为基于计算机系统的一种元素；在软件需求分析时，使用“问题环境中熟悉的”术语来描述软件解决方案；当我们在设计过程中时，降低了抽象级别，最终，当源代码生成时，就到达了最低的抽象级别。



## ② 模块化

计算机软件的模块化概念已经被采纳了将近40年，软件体系结构体现了模块化，即软件被划分成独立命名和可独立访问的被称作模块的构件，它们被集成到一起满足问题需求。

在考虑模块化时，需要关注以下几个方面的问题。

- 模块可分解性：如果一种设计方法提供了将问题分解成子问题的系统化机制，它就能降低整个系统的复杂性，从而实现一种有效的模块化解决方案。
- 模块可组装性：如果一种设计方法使现存的（可复用的）设计构件能被组装成新系统，它就能提供一种不一切从头开始的模块化解决方案。
- 模块可理解性：如果一个模块可以作为一个独立的单位（不用参考其他模块）被理解，那么它就易于构造和修改。
- 模块连续性：如果对系统需求的微小修改只导致对单个模块，而不是整个系统的修改，则修改引起的副作用就会被最小化。
- 模块保护：如果模块内出现异常情况，并且它的影响限制在模块内部，则错误引起的副作用就会被最小化。

最后，要注意的是：即使系统的实现必须是“单块集成电路式的”，它也可以采用模块化的设计。有些情况（例如，实时软件、嵌入式软件）下，由于程序（例如，子例程、过程）造成的即使是比较小的速度和内存开销也是无法接收的。在这种情况下，软件能够而且也应该采用模块化设计作为一种指导思想，可以“内嵌地（in line）”开发代码。尽管程序源代码看上去可能没有模块性，但这种思想依然有效，而且程序对提供模块化系统非常有益。

## ③ 信息隐蔽

隐蔽的含义是：有效的模块化可以通过定义一组独立模块来实现，这些模块相互之间只交流实现软件功能必需的信息。抽象有助于定义组成软件的过程（或信息）实体，隐蔽定义并加强了对模块内部过程细节或模块使用的任何局部数据结构的访问约束。

模块的信息隐藏可以通过接口设计来实现。一个模块仅提供有限个接口（interface），对象模块的功能或与模块交流信息必须且只需通过调用公有接口来实现。举个例子，如果模块是一个C++对象，那么该模块的公有接口就对应于对象的公有函数。如果模块是一个COM对象，那么模块的公有接口就是COM对象的接口。一个COM对象可以有多个接口，而每个接口实际上是一些函数的集合。

将信息隐藏用做模块化系统的设计标准为在测试及以后维护中需做修改时提供了极大的方便，因为大多数数据和过程对软件其他部分是隐蔽的，修改时无意中引入的错误不太可能传播到软件中其他位置。

## ④ 内聚和耦合

内聚 (cohesion) 是一个模块内部各成分之间相关联程度的度量。耦合 (coupling) 是模块之间依赖程度的度量。内聚和耦合是密切相关的, 与其他模块存在强耦合的模块通常意味着弱内聚, 而强内聚的模块通常意味着与其他模块之间存在弱耦合。模块设计追求强内聚, 弱耦合。内聚和耦合都有强度上的区别, 这里做个比较。

内聚按强度从低到高有以下几种类型。

- 偶然内聚: 如果一个模块的各成分之间毫无关系, 则称为偶然内聚。
- 逻辑内聚: 几个逻辑上相关的功能被放在同一模块中, 则称为逻辑内聚, 如一个模块读取各种不同类型外设的输入。尽管逻辑内聚比偶然内聚合理一些, 但逻辑内聚的模块各成分在功能上并无关系, 即使局部功能的修改有时也会影响全局, 所以这类模块的修改也比较困难。
- 时间内聚: 如果一个模块完成的功能必须在同一时间内执行 (如系统初始化), 但这些功能只是因为时间因素关联在一起, 则称为时间内聚。
- 过程内聚: 如果一个模块内部的处理成分是相关的, 而且这些处理必须以特定的次序执行, 则称为过程内聚。
- 通信内聚: 如果一个模块的所有成分都操作同一数据集或生成同一数据集, 则称为通信内聚。
- 顺序内聚: 如果一个模块的各个成分和同一个功能密切相关, 而且一个成分的输出作为另一个成分的输入, 则称为顺序内聚。
- 功能内聚: 模块的所有成分对于完成单一的功能都是必需的, 则称为功能内聚。

耦合的强度依赖于以下几个因素: 一个模块对另一个模块的调用; 一个模块向另一个模块传递的数据量; 一个模块施加到另一个模块的控制的多少; 模块之间接口的复杂程度。

耦合按从强到弱的顺序可分为以下几种类型。

- 内容耦合: 当一个模块直接修改或操作另一个模块的数据, 或者直接转入另一个模块时, 就发生了内容耦合。此时, 被修改的模块完全依赖于修改它的模块。
- 公共耦合: 两个以上的模块共同引用一个全局数据项就称为公共耦合。
- 控制耦合: 一个模块在界面上传递一个信号 (如开关值、标志量等) 控制另一个模块, 接收信号的模块的动作根据信号值进行调整, 称为控制耦合。
- 标记耦合: 模块间通过参数传递复杂的内部数据结构, 称为标记耦合。此数据结构的变化将使相关的模块发生变化。
- 数据耦合: 模块间通过参数传递基本类型的数据, 称为数据耦合。
- 非直接耦合: 模块间没有信息传递时, 属于非直接耦合。

如果模块间必须存在耦合, 就尽量使用数据耦合, 少用控制耦合, 限制公共耦合的范围, 坚决避免使用内容耦合。

### ⑤ 数据结构与算法设计

学会设计数据结构与算法，可以让我们编写出高效率的程序。设计高效率的程序是基于良好的数据结构与算法的，而不是基于编程小技巧。大多数计算机科学系在设置课程时，都重视学习基本的软件工程原理，以及数据结构与算法设计。

一般说来，数据结构与算法就是一类数据的表示及其相关的操作（这里算法不是指数值计算的算法）。从数据表示的观点来看，存储在数组中的一个有序整数表也是一种数据结构。算法是指对数据结构施加的一些操作，例如对一个线性表进行检索、插入、删除等操作。一个算法如果能在所要求的资源限制（resource constraints）范围内将问题解决好，则称这个算法是有效率（efficient）的。例如，一个资源限制可能是“用于存储数据的内存有限”，或者“允许执行每个子任务所需的时间有限”。一个算法如果比其他已知算法所需要的资源都少，这个算法也被称为是有效率的。算法的代价（cost）是指消耗的资源量。一般说来，代价是由一个关键资源例如时间或空间来评估的。

毋庸置疑，人们编写程序是为了解决问题。只有通过预先分析问题来确定必须达到的性能目标，才有希望挑选出正确的数据结构。有相当多的程序员忽视了这一分析过程，而直接选用某一个他们习惯使用的，但是与问题不相称的数据结构，结果设计出了一个低效率的程序。如果使用简单的设计就能够达到性能目标时，选用复杂的数据结构也是没有道理的。

人们对常用的数据结构与算法的研究已经相当透彻，可以归纳出一些设计原则。

- 每一种数据结构与算法都有其时间、空间的开销和收益。当面临一个新的设计问题时，设计者要彻底掌握怎样权衡时空开销和算法有效性的方法。这就需要懂得算法分析的原理，而且还需要了解所使用的物理介质的特性（例如，数据存储在磁盘上与存储在内存中就有不同的考虑）。
- 与开销和收益有关的是时间-空间的权衡。通常可以用更大的时间开销来换取空间的收益，反之亦然。时间-空间的权衡普遍地存在于软件开发的各个阶段中。
- 程序员应该充分地了解一些常用的数据结构与算法，避免不必要的重复设计工作。
- 数据结构与算法为应用服务。我们必须先了解应用的需求，再寻找或设计与实际应用相匹配的数据结构。

### (3) 系统架构设计风格

软件体系结构设计的一个核心问题是能否使用重复的体系结构模式，即能否达到体系结构级的软件重用。也就是说，能否在不同的软件系统中，使用同一体系结构。基于这个目的，学者们开始研究和实践软件体系结构的风格和类型问题。

软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式，它反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一

个完整的系统。按这种方式理解,软件体系结构风格定义了用于描述系统的术语表和一组指导构件系统的规则。

对软件体系结构风格的研究和实践促进了对设计的复用,一些经过实践证实的解决方案也可以可靠地用于解决新的问题。体系结构风格的不变部分使不同的系统可以共享同一个实现代码。只要系统是使用常用的、规范的方法来组织,就可使别的设计者很容易地理解系统的体系结构。例如,如果某人把系统描述为客户/服务器模式,则不必给出设计细节,我们立刻就会明白系统是如何组织和工作的。

限于篇幅,在本节中,我们将只介绍几种主要的和经典的体系结构风格以及它们的优缺点。

### ① 连接件风格

可以概括为:通过连接件绑定在一起的、按照一组规则运作的并行构件网络。连接件风格中的系统组织规则如下:

- 系统中的构件和连接件都有一个顶部和一个底部。
- 构件的顶部应连接到某连接件的底部,构件的底部则应连接到某连接件的顶部,而构件与构件之间的直接连接是不允许的。
- 一个连接件可以同任意数目的其他构件和连接件连接。
- 当两个连接件进行直接连接时,必须由其中一个的底部到另一个的顶部。

图 5-6 所示为连接件风格的示意图。图中构件与连接件之间的连接体现了连接件风格中构建系统的规则。

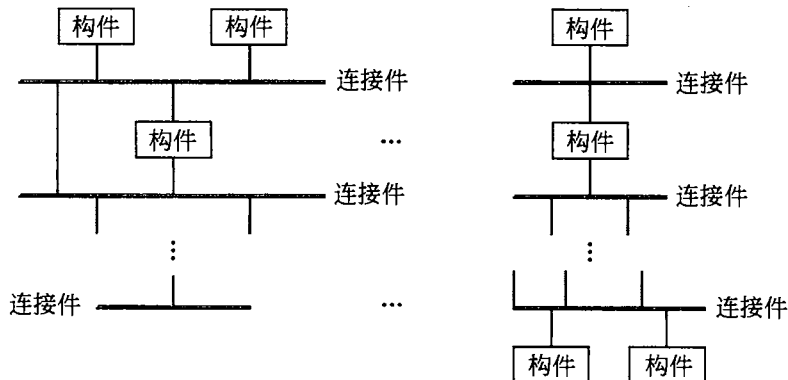


图 5-6 连接件风格的体系结构

连接件风格是最常用的一种软件体系结构风格。从连接件风格的组织规则和结构图中,我们可以看出,连接件风格具有以下特点。

- 系统中的构件可实现应用需求，并能将任意复杂度的功能封装在一起。
- 所有构件之间的通信是通过以连接件为中介的异步消息交换机制来实现的。
- 构件相对独立，构件之间依赖性较少。系统中不存在某些构件将在同一地址空间内执行，或某些构件共享特定控制线程之类的相关性假设。

## ② 管道/过滤器风格

在管道/过滤器风格的软件体系结构中，每个构件都有一组输入和输出，构件读输入的数据流，经过内部处理，然后产生输出数据流。这个过程通常通过对输入流的变换及增量计算来完成，所以在输入被完全使用之前，输出便产生了。因此，这里的构件被称为过滤器，这种风格的连接件就像是数据流传输的管道，将一个过滤器的输出传到另一过滤器的输入。此风格特别重要的过滤器必须是独立的实体，它不能与其他的过滤器共享数据，而且一个过滤器不知道它上游和下游的标识。一个管道/过滤器网络输出的正确性并不依赖于过滤器进行增量计算过程的顺序。

图 5-7 所示为管道/过滤器风格的示意图。一个典型的管道/过滤器体系结构的例子是用 UNIX Shell 编写的程序。UNIX 既提供一种符号，以连接各组成部分（UNIX 的进程），又提供某种进程运行时机制以实现管道。另一个著名的例子是传统的编译器。传统的编译器一直被认为是一种管道系统，在该系统中，一个阶段（包括词法分析、语法分析、语义分析和代码生成）的输出是另一个阶段的输入。

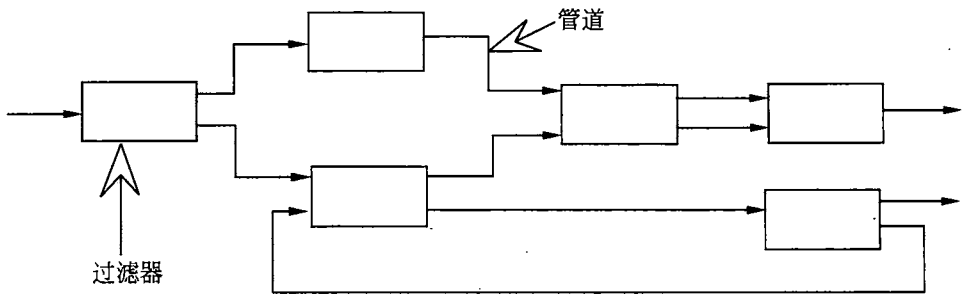


图 5-7 管道/过滤器风格的体系结构

管道/过滤器风格的软件体系结构具有许多很好的特点：

- 使得软件构件具有良好的隐蔽性和高内聚、低耦合的特点。
- 允许设计者将整个系统的 I/O 行为看成是多个过滤器的简单合成。
- 支持软件重用。主要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来。
- 易于系统维护和增强系统性能。新的过滤器可以添加到现有系统中来，旧的可以

被改进的过滤器替换掉。

- 允许对一些如吞吐量、死锁等属性的分析。
- 支持并行执行。每个过滤器是作为一个单独的任务完成，所以可与其他任务并行执行。

但是，这样的系统也存在着若干不利因素。

- 通常导致进程成为批处理的结构。这是因为虽然过滤器可增量式地处理数据，但它们是独立的，所以设计者必须将每个过滤器看成一个完整的从输入到输出的转换。
- 不适合处理交互的应用。当需要增量地显示改变时，这个问题尤为严重。
- 因为在数据传输上没有通用的标准，每个过滤器都增加了解析和合成数据的工作，这样就导致了系统性能下降，并增加了编写过滤器的复杂性。

### ③ 数据抽象和面向对象风格

抽象数据类型概念对软件系统有着重要作用，目前软件界已普遍转向使用面向对象系统。这种风格建立在数据抽象和面向对象的基础上，数据的表示方法和它们的相应操作封装在一个抽象数据类型或对象中。这种风格的构件是对象，或者说是抽象数据类型的实例。对象是一种被称作管理者的构件，因为它负责保持资源的完整性。对象是通过函数和过程的调用来交互的。

图 5-8 所示为数据抽象和面向对象风格的示意图。

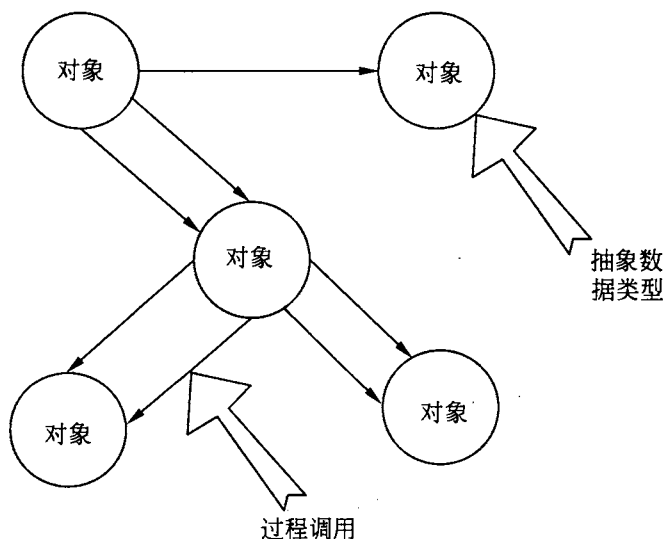


图 5-8 数据抽象和面向对象风格的体系结构





面向对象的系统有许多的优点，并早已为人所知：

- 因为对象对其他对象隐藏它的表示，所以可以改变一个对象的表示，而不影响其他的对象。
- 设计者可将一些数据存取操作的问题分解成一些交互的代理程序的集合。

但是，面向对象的系统也存在着某些问题：

- 为了使一个对象和另一个对象通过过程调用等进行交互，必须知道对象的标识。只要一个对象的标识改变了，就必须修改所有其他明确调用它的对象。
- 必须修改所有显式调用它的其他对象，并消除由此带来的一些副作用。例如，如果 A 使用了对象 B，C 也使用了对象 B，那么，C 对 B 的使用所造成的对 A 的影响可能是意想不到的。

#### ④ 基于事件的隐式调用风格

基于事件的隐式调用风格的思想是构件不直接调用一个过程，而是触发或广播一个或多个事件。系统中的其他构件中的过程在一个或多个事件中注册，当一个事件被触发，系统自动调用在这个事件中注册的所有过程，这样，一个事件的触发就导致了另一模块中的过程的调用。

从体系结构上说，这种风格的构件是一些模块，这些模块既可以是一些过程，又可以是一些事件的集合。过程可以用通用的方式调用，也可以在系统事件中注册一些过程，当发生这些事件时，过程被调用。

基于事件的隐式调用风格的主要特点是：事件的触发者并不知道哪些构件会被这些事件影响。这样不能假定构件的处理顺序，甚至不知道哪些过程会被调用，因此，许多隐式调用的系统也包含显式调用作为构件交互的补充形式。

支持基于事件的隐式调用的应用系统很多。例如，在编程环境中用于集成各种工具、在数据库管理系统中确保数据的一致性约束、在用户界面系统中管理数据，以及在编辑器中支持语法检查。例如，在某系统中，编辑器和变量监视器可以登记相应调试器的断点事件。当调试器在断点处停下时，它声明该事件，由系统自动调用处理程序，如编辑程序可以卷屏到断点，变量监视器刷新变量数值。而调试器本身只声明事件，并不关心哪些过程会启动，也不关心这些过程做什么处理。

隐式调用系统的主要优点有：

- 为软件重用提供了强大的支持。当需要将一个构件加入现存系统中时，只需将它注册到系统的事件中。
- 为改进系统带来了方便。当用一个构件代替另一个构件时，不会影响到其他构件的接口。

隐式调用系统的主要缺点有：

- 构件放弃了对系统计算的控制。一个构件触发一个事件时，不能确定其他构件是否会响应它。而且即使它知道事件注册了哪些构件的构成，它也不能保证这些过程被调用的顺序。
- 数据交换的问题。有时数据可被一个事件传递，但另一些情况下，基于事件的系统必须依靠一个共享的仓库进行交互。在这些情况下，全局性能和资源管理便成了问题。
- 既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理也就存在问题。

#### ⑤ 层次系统风格

层次系统组织成一个层次结构，每一层为上层服务，并作为下层客户。在一些层次系统中，除了一些精心挑选的输出函数外，内部的层只对相邻的层可见。这样的系统中构件在一些层实现了虚拟机（在另一些层次系统中层是部分不透明的）。连接件通过决定层间如何交互的协议来定义，拓扑约束包括对相邻层间交互的约束。

这种风格支持基于可增加抽象层的设计。这样，允许将一个复杂问题分解成一个增量步骤序列的实现。由于每一层最多只影响两层，同时只要给相邻层提供相同的接口，所以允许每层用不同的方法实现，同样为软件重用提供了强大的支持。

图 5-9 所示为层次系统风格的示意图。层次系统最广泛的应用是分层通信协议。在这一应用领域中，每一层提供一个抽象的功能，作为上层通信的基础。较低的层次定义底层的交互，最底层通常只定义硬件物理连接。

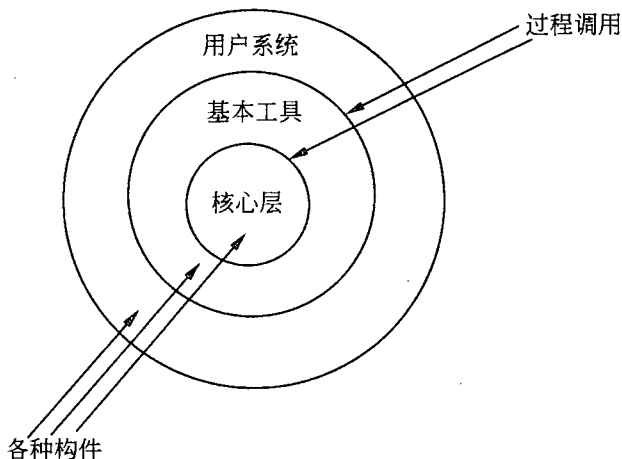


图 5-9 层次系统风格的体系结构

层次系统有许多可取的属性：

- 支持基于抽象程度递增的系统设计，使设计者可以把一个复杂系统按递增的步骤进行分解。
- 支持功能增强，因为每一层至多和相邻的上下层交互，所以功能的改变最多影响相邻的上下层。
- 支持重用。只要提供的服务接口定义不变，同一层的不同实现可以交换使用。这样，就可以定义一组标准的接口，而允许各种不同的实现方法。

但是，层次系统也有其不足之处：

- 并不是每个系统都可以很容易地划分为分层的模式，甚至即使一个系统的逻辑结构是层次化的，出于对系统性能的考虑，系统设计师不得不把一些低级或高级的功能综合起来。
- 很难找到一个合适的、正确的层次抽象方法。

## 5.4.2 系统详细设计

### 1. 系统详细设计在软件生命周期中的作用

设计阶段由架构设计（有时也称为概要设计）和详细设计组成。架构设计是高层设计，其任务是定义子系统（又可以成为包），包括子系统间的依赖关系和主要通信机制。子系统有利于描述系统的逻辑组成部分以及各部分之间的依赖关系。详细设计就是要细化子系统的内容，清晰描述子系统的内部结构，同时使用动态模型描述在特定环境下子系统的内部行为。图 5-10 可以说明两者之间的关系。

详细设计阶段的任务就是把子系统具体化。详细设计主要是针对程序开发部分来说的。但这个阶段不是真正编写程序，而是设计出程序的详细规格说明。这种规格说明的作用很类似于其他工程领域中工程师经常使用的工程蓝图，它们应该包含必要的细节，例如，程序界面、表单、需要的数据等。程序员可以根据它们写出实际的程序代码。

### 2. 系统详细设计阶段用到的设计方法简介

在系统详细设计阶段用到的设计方法总体上可以分为面向过程的设计方法和面向对象的设计方法，因为这两种方法都包括了太多的内容，它们所涉及的知识超出了本节范围，所以只在这里做一下简单介绍。

#### （1）过程式设计

总的来说，过程式的程序设计是一种自上而下的设计方法，设计者用一个处于最顶层的过程概括出整个应用程序需要做的事，而这个最顶层的过程由一系列子过程的调用组成。对于最顶层的过程中的每一个子过程，都又可以再被精练成更小的子过程。重复这个过程，就可以完成一个过程式的设计。其特征是以过程为中心，用过程来作为划分程序的基本单位，数据在过程式设计中往往处于从属的位置。

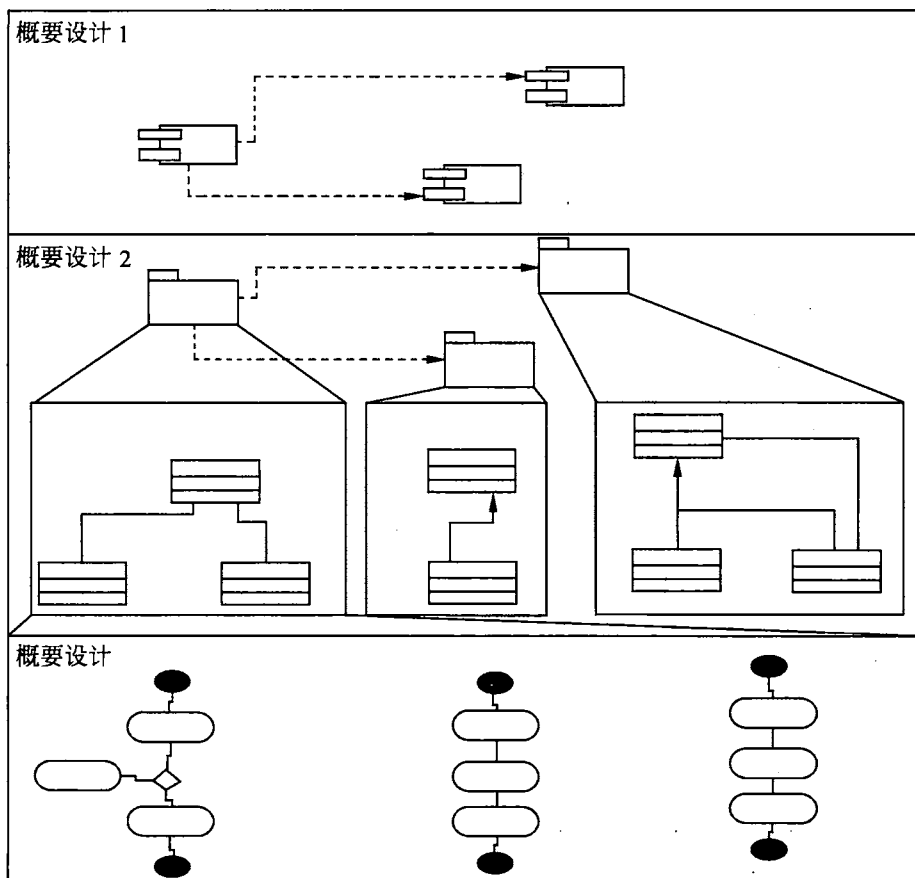


图 5-10 架构设计和详细设计间的关系

过程式设计的优点是易于理解和掌握，这种逐步细化问题的设计方法和大多数人的思维方式比较接近。

然而，过程式设计对于比较复杂的问题，或是在开发中需求变化比较多时，往往显得力不从心。这是因为过程式的设计是自上而下的，这要求设计者在一开始就要对需要解决的问题有一定的了解。在问题比较复杂时，要做到这一点会比较困难，而当开发中需求变化的时候，以前对问题的理解也许会变得不再适用。事实上，开发一个系统的过程往往也是一个对系统不断了解和学习的过程，而过程式的设计方法忽略了这一点。

在过程式设计的语言中，一般都既有定义数据的元素，如 C 语言中的结构，也有定义操作的元素，如 C 语言中的函数。这样做的结果是数据和操作被分离开，容易导致对一种数据的操作分布在整个程序的各个角落，而一个操作也可能会用到很多种数据，在这种情

况下，对数据和操作的任何一部分进行修改都会变得很困难。

在过程式设计中，最顶层的过程处于一个非常重要的地位。设计者正是在最顶层的过程中，对整个系统进行一个概括的描述，再以此为起点，逐步细化出整个系统。然而，这样做的一个后果，是容易将一些较外延和易变化的逻辑（比如用户交互）同程序的核心逻辑混淆在一起。假设我们编写一个图形界面的计算器程序和一个命令行界面的计算器程序，可以想象这两个版本的顶层过程会有很大的差异，由此衍生出来的程序很有可能也会迥然不同，而这两个版本的程序本应该有很大部分可以共用才对。

过程式设计还有一个问题就是其程序架构的依赖关系问题。以 C 语言为例，一个典型的过程式程序往往如图 5-11 所示。

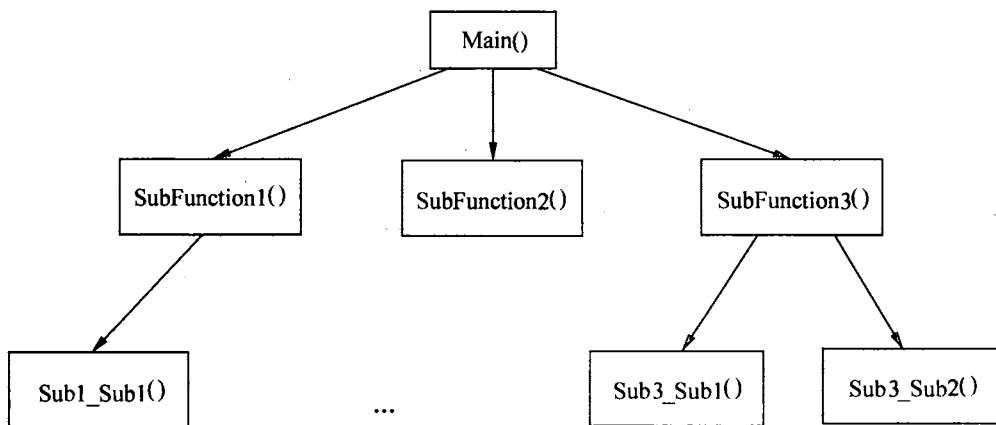


图 5-11 一个典型的过程式程序

图 5-11 中的箭头代表了函数间的调用关系，也是函数间的依赖关系。如图 5-11 所示，main() 函数依赖于其子函数，这些子函数又依赖于更小的子函数，而在程序中，越小的函数处理的往往是细节实现，这些具体的实现，又常常变化。这样的结果，就是程序的核心逻辑依赖于外延的细节，程序中本来应该都是比较稳定的核心逻辑，也因为依赖于易变化的部分，而变得不稳定起来，一个细节上的小小改动，也有可能依赖关系上引发一系列变动。可以说这种依赖关系也是程序设计不能很好处理变化的原因之一，而一个合理的依赖关系，应该是由细节实现依赖于核心逻辑才对。

## (2) 面向对象设计

面向对象是一种自下而上的程序设计方法。不像过程式设计那样一开始就要用顶层过程概括出整个系统行为，面向对象设计往往从问题的一部分着手，一点点地构建出整个程序。面向对象设计以数据为中心，类作为表现数据的工具，是划分程序的基本单位，而过

程在面向对象设计中成为了类的接口。

面向对象设计自下而上的特性，允许开发者从问题的局部开始，在开发过程中逐步加深对系统的理解。这些新的理解以及开发中遇到的需求变化，都会再作用到系统开发本身，形成一种螺旋式的开发方式（在这种开发方式中，对于已有的代码，常需要运用重构技术来做代码重构以体现系统的变化）。

同过程相比，数据应该是程序中更稳定的部分，比如，一个网上购物程序，无论怎么变化，大概都会处理货物、客户这些数据对象。不过在这里，只有从抽象的角度来看，数据才是稳定的，如果考虑这些数据对象的具体实现，它们甚至比函数还要不稳定，因为在一个数据对象中增减字段在程序开发中是常事。因此，在以数据为中心构建程序的同时，我们需要一种手段来抽象地描述数据，这种手段就是使用函数。在面向对象设计中，类封装了数据，而类的成员函数作为其对外的接口，抽象地描述了类。用类将数据和操作这些数据的函数放在一起，这可以说就是面向对象设计方法的本质。

在面向对象设计中，类之间的关系有两种：客户（client）关系和继承（inheritance）关系。客户关系如图 5-12 所示，表示一个类（Client）会使用到另一个类（Server）。一般将这种关系中的 Client 类称为客户端，Server 类称为服务器。

继承关系如图 5-13 所示，表示一个类（Child）对另一个类（Parent）的继承。一般将这种关系中的 Parent 类称为父类，Child 类称为子类。

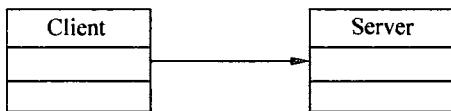


图 5-12 客户关系

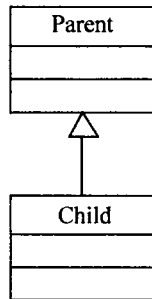


图 5-13 继承关系

面向对象设计的过程就是将各个类按以上的两种关系组合在一起，这两种关系都非常简单，不过组合在一起却能提供强大的设计能力。

### 5.4.3 系统测试

#### 1. 系统测试在软件生命周期中的作用

软件测试的重要性及其对软件质量的好坏的影响是非常重要的。

软件系统的开发包括一系列生产活动，其中由人带来的错误因素非常多。错误可能出



现在程序的最初,当时目标可能是错误的或描述不完整,也可能在后期的设计和开发阶段,因为人们不能完好无缺地工作和交流。软件开发过程中必须伴有质量保证活动。

软件测试是软件质量保证的关键元素,代表了规约、设计和编码的最终检查。

不断增加的软件故障带来的过高代价使得人们注重于规划良好的测试活动,软件开发组织将 30%~40% 的项目精力花在测试上并不为怪。另一方面,人命悠关的软件(如飞行控制和核反应堆)测试所花的时间往往是其他软件工程活动时间之和的三到五倍。

对于软件测试的概念,经常会有这样的误解:

- 软件测试仅仅是对程序的测试。
- 软件测试是软件开发的一个独立阶段,只有在软件开发的测试阶段才会来做测试的工作。

根据软件定义,软件包括程序、数据和文档,所以软件测试并不仅仅是程序测试,并且软件测试不仅仅是软件开发的一个独立阶段,而应贯穿于整个软件生命周期中。实际上,当确定系统范围后,就可以在初始阶段制定初始的测试计划,因为测试主要应用在对每个构造(系统实现的结果)进行集成测试和系统测试。这也意味着测试即是细化阶段的焦点,也是构造阶段的焦点。在移交阶段,焦点转向修复在早期使用中发现的缺陷,并进行回归测试,如图 5-14 所示。

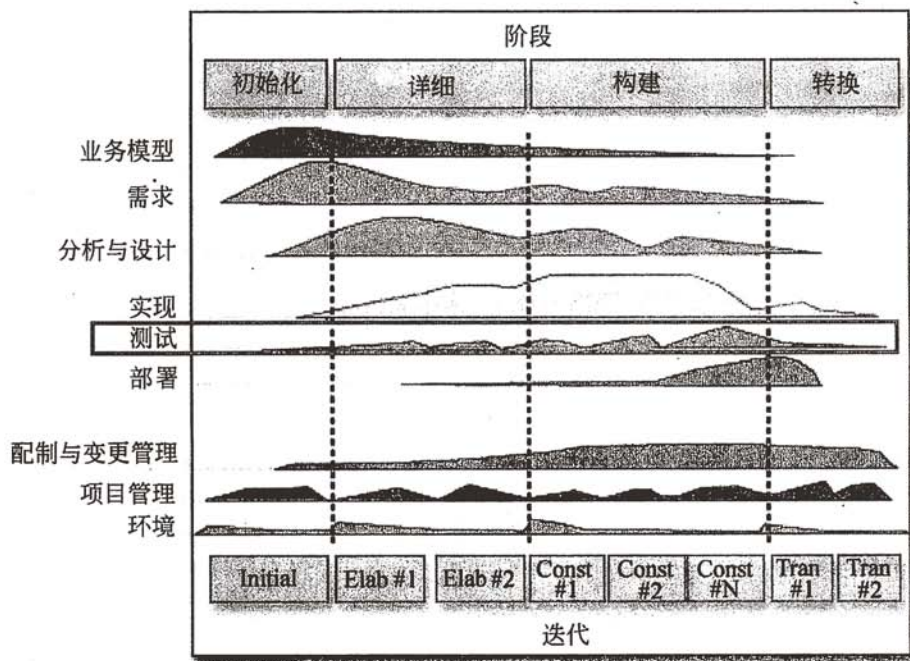


图 5-14 系统测试

在整个软件生命周期中，各阶段有不同的测试对象，形成了不同开发阶段的不同类型的测试。需求分析、概要设计、详细设计以及程序编码等各阶段所得到的文档，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都应成为软件测试的对象。

- 在软件编码结束后，对编写的每一个程序模块进行测试，称为“模块测试”或“单元测试”。
- 在模块集成后，对集成在一起的模块组件，有时也可称为“部件”进行测试，称为集成测试。
- 在集成测试后，需要检测与证实软件是否满足软件需求说明书中规定的要求，这就称为确认测试。
- 将整个程序模块集成为软件系统安装在运行环境下，对于硬件、网络、操作系统及支撑平台构成的整体系统进行测试，称为系统测试。

本节主要讨论软件测试中的系统测试的一些基本概念和策略。

## 2. 系统测试类型

系统测试事实上是对整个基于计算机的系统进行考验的一系列不同测试。虽然每一个测试都有不同的目的，但所有都是为了整个系统成分能正常地集成到一起以完成分配的功能而工作的。这里我们讨论对基于计算机系统有用的系统测试类型。

### (1) 恢复测试

许多基于计算机的系统必须在一定的时间内从错误中恢复过来，然后继续运行。在有些情况下，一个系统必须是可以容错的，这就是说，运行过程中的错误必须不能使得整个系统的功能都停止。在其他情况下，一个系统错误必须在一个特定的时间段之内改正，否则就会产生严重的经济损失。

恢复测试是通过各种手段，让软件强制性地发生故障，然后来验证恢复是否能正常进行的一种系统测试方法。如果恢复是自动的(由系统本身来进行的)，重新初始化、检查点机制、数据恢复和重新启动都要进行正确验证；如果恢复是需要人工干预的，那么要估算修复的平均时间是否在可以接受的范围之内。

### (2) 安全测试

任何管理敏感信息或者能够对个人造成不正当伤害的计算机系统都是不正当的或非法侵入的目标。侵入包括了范围很广的活动：只是为练习而试图侵入系统的黑客；为了报复而试图攻破系统的有怨言的雇员；还有为了得到非法的利益而试图侵入系统的不诚实的个人。

安全测试用来验证集成在系统内的保护机制是否能够在实际中保护系统不受到非法侵入。“系统的安全当然必须能够经受住正面的攻击，但是它也必须能够经受住侧面的和背后的攻击。”

在安全测试过程中，测试者扮演着一个试图攻击系统的个人角色。测试者可以尝试去



通过外部的手段来获取系统的密码，使用可以瓦解任何防守的客户软件来攻击系统；可以把系统“制服”，使得别人无法访问；可以有目的地引发系统错误，期望在系统恢复过程中侵入系统；可以通过浏览非保密的数据，从中找到进入系统的钥匙，等等。

只要有足够的时间和资源，好的安全测试就一定能够最终侵入一个系统。系统设计者的任务就是要把系统设计为想要攻破系统而付出的代价大于攻破系统之后得到的信息的价值。

### （3）压力测试

在较早的软件测试步骤中，白盒和黑盒技术对正常的程序功能和性能进行了详尽的检查。压力测试（stress testing）的目的是要对付非正常的情形。在本质上说，进行压力测试的人应该这样问：“我们能够将系统折腾到什么程度而又不会出错？”

压力测试是在一种需要反常数量、频率或资源的方式下执行系统。例如，当平均每秒出现一个或两个中断的情况下，应当对每秒出现 10 个中断的情形来进行特殊的测试；把输入数据的量提高一个数量级来测试输入功能会如何响应；应当执行需要最大的内存或其他资源的测试实例；使用在一个虚拟的操作系统中会引起颠簸的测试实例；可能会引起大量的驻留磁盘数据的测试实例。从本质上来说，测试者是想要破坏程序。

压力测试的一个变种是一种被称为是敏感测试的技术。在有些情况（最常见的是在数学算法中）下，在有效数据界限之内的一个很小范围的数据可能会引起极端的甚至是错误的运行，或者引起性能的急剧下降，这种情况同数学函数中的奇点相类似。敏感测试就是要发现在有效数据输入里的可能会引发不稳定或者错误处理的数据组合。

### （4）性能测试

在实时系统和嵌入系统中，符合功能需求但不符合性能需求的软件是不能接受的。性能测试就是用来测试软件在集成系统中的运行性能的。性能测试可以发生在测试过程的所有步骤中，即使是在单元层，一个单独模块的性能也可以使用白盒测试来进行评估。然而，只有当整个系统的所有成分都集成到一起之后，才能检查一个系统的真正性能。

性能测试经常和压力测试一起进行，而且常常需要硬件和软件测试设备。这就是说，在一种苛刻的环境中衡量资源的使用（比如，处理器周期）常常是必要的。外部的测试设备可以监测执行的间歇，当出现情况（比如中断）时记录下来。通过对系统的检测，测试者可以发现导致效率降低和系统故障的情况。

因为前面提到了白盒测试和黑盒测试，在这里简单介绍一下。

白盒测试和黑盒测试区别在于测试是否针对着系统的内部结构和具体实现算法。

#### ① 黑盒测试

黑盒测试也称功能测试或数据驱动测试，它是在已知产品所应具有的功能的情况下，通过测试来检测每个功能是否都能正常使用。在测试时，把程序看做一个不能打开的黑盒子，在完全不考虑程序内部结构和内部特性的情况下，测试者在程序接口进行测试，它只

检查程序功能是否按照需求规格说明书的规定正常使用,程序是否能适当地接收输入数据而产生正确的输出信息,并且保持外部信息(如数据库或文件)的完整性。黑盒测试方法主要有等价类划分、边值分析、因果图、错误推测等,主要用于软件确认测试。黑盒法着眼于程序外部结构,不考虑内部逻辑结构、针对软件界面和软件功能进行测试。黑盒法是穷举输入测试,只有把所有可能的输入都作为测试情况使用,才能以这种方法查出程序中所有的错误。实际上测试情况有无穷多个,人们不仅要测试所有合法的输入,而且还要对那些不合法但是可能的输入进行测试。

### ② 白盒测试

白盒测试也称结构测试或逻辑驱动测试,它是在知道产品内部工作过程的情况下,通过测试来检测产品内部动作是否按照规格说明书的规定正常进行;按照程序内部的结构测试程序,检验程序中的每条通路是否都有能按预定要求正确工作。白盒测试的主要方法有逻辑驱动、基路测试等,主要用于软件验证。

白盒法全面了解程序内部逻辑结构、对所有逻辑路径进行测试。白盒法是穷举路径测试。在使用这一方案时,测试者必须检查程序的内部结构,从检查程序的逻辑着手,得出测试数据。贯穿程序的独立路径数是天文数字。但即使每条路径都测试了仍然可能有错误:第一,穷举路径测试绝不能查出程序违反了设计规范,即程序本身是个错误的程序;第二,穷举路径测试不可能查出程序中因遗漏路径而出错;第三,穷举路径测试可能发现不了一些与数据相关的错误。

### 3. 系统测试的策略

高质量的测试工作才能对系统质量的提高起到实质性的作用,而测试的质量会由测试时所采取的策略来决定。在制定系统测试的策略时,有一些原则和需要注意的问题。

在着手开始测试之前的较长时间内,就要以量化的形式确定产品的需求。虽然测试的主要目的是找错误,一个好的测试策略同样可以评估其他的质量特性,比如可移植性、可维护性和可用性。这些应当用一种可以测度的方式来表示,从而保证测试结果是不含糊的。

明显地指出测试目标,测试的特定目标应当用可以测度的术语来描述。比如,测试有效性、测试覆盖率、故障出现的平均时间、发现和改正缺陷的开销、允许剩余的缺陷密度或出现频率,以及每次回归测试的工作时间都应当在测试计划中清楚的说明。

了解软件的用户并为每一类用户建立相应档案,通过着重于测试产品的实际用途,对使用实例(描述每一类用户在实际使用系统时同系统的交互情况)构成的图进行研究,可以减少整个测试的工作量。

设计一个能够测试自身是否“强壮”的软件。这就是说,软件应当能够诊断特定类型的错误,另外,设计应当能够包括自动测试和回归测试。

使用有效的正式技术复审作为测试之前的过滤器。正式技术复审在发现错误方面可以

同测试一样有效，由于这个原因，复审可以减少为了得到高质量软件所需的测试工作量。

使用正式技术复审来评估测试策略和测试用例本身。正式的技术复审可以发现在测试过程中的不一致性、遗漏和完全的错误。这样可以节省时间，同时也能够提高产品的质量。

为测试过程建立一种连续改善的实现方法。测试策略必须进行测量，在测试过程中收集的度量数据应当被用做软件测试的统计过程控制方法的一部分。

这里给出一个在实际工作测试中采用的策略清单来结束本节：

测试目的：

- 测试的范围，哪些功能要包括在内，哪些要排除在外。
- 谁是客户和最终用户，谁就是测试结果的验收者。
- 测试的次序和日程安排。
- 验收的条件、成功因素、限制。

资源需求：

- 制定计划和运行测试需要哪些技术和经验。
- 相关人员的角色和责任。
- 谁将对测试工作进行全盘协调。
- 谁负责测试资料管理、版本控制、错误跟踪。

测试环境：

- 用于测试的系统配置怎样。
- 需要什么自动化工具。
- 需要哪些测试数据（数据库和输入交易），以及如何安装。
- 如何前调系统时钟。

测试过程：

- 运行测试时要遵循哪些过程（设置、执行、记录）。
- 测试案例如何制定，其标准格式是什么。
- 测试案例定义的覆盖要求是什么。
- 遇到问题如何确定其严重程度，对问题如何处理。

## 5.5 系统维护知识

### 5.5.1 系统运行管理

#### 1. 运行管理制度

一个规范管理的企业，首先就要具备和管理相关的各种规章制度，只有这样才能做到

有章可循。我们知道在通常的情况下，每一项具体的业务都有一套科学的运行制度。信息系统也不例外，同样也需要一套管理制度，以确保信息系统的正常和安全运行。

信息系统的管理制度主要分为各类机房安全运行的管理制度和其他管理制度。我们首先来看一下机房的的安全管理制度。

#### （1）各类机房安全运行管理制度

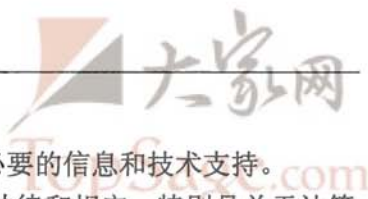
信息系统的运行制度，首先表现为物理意义上的机房必须处于监控之中。机房安全运行制度应该包括如下内容：

- 身份登记制度与人员出入验证制度。
- 人员携带物品，尤其是大型物品离开机房时的审查登记制度。
- 当有客人需要参观机房的时候，需要严格的身份审查制度。
- 每天都需要配备专人负责启动和关闭计算机系统。
- 对系统运行状况进行监视，跟踪并详细记录其所运行信息。
- 对系统及相关的硬件设备进行定期保养和维护。
- 操作人员必须在指定的计算机或终端上进行操作，同时还要对操作的内容进行规范和管理。
- 不做与工作无关的操作，不下载、不运行来历不明的软件。
- 不越权运行程序，不查阅无关参数。
- 当遇到操作异常的情况，立即向相关部门和人员报告。

#### （2）信息系统相关的其他管理制度

信息系统的运行制度，还表现为软件、数据、信息等其他要素必须处于监控之中。信息系统的其他管理制度主要包括：

- 必须有重要的系统软件、应用软件管理制度，如系统软件的更新维护、应用软件的源程序与目标程序分离等。
- 必须有数据管理制度。例如，重要输入数据、输出数据的管理。
- 必须有密码口令管理制度，做到口令专管专用，定期更改并在失密后立即报告。
- 必须有网络通信安全管理制度，实行网络电子公告系统的用户登记和对外信息交流的管理制度。
- 必须有病毒的防治管理制度，及时检测、清除计算机病毒，并备有检测、清除情况的记录。
- 必须有人员调离的安全管理制度。例如，人员调离的同时马上收回钥匙、移交工作、更换口令、取消账号，并向被调离的工作人员说明其保密义务。
- 建立安全培训制度，进行计算机安全法律教育、职业道德教育和计算机安全技术



教育。对关键岗位的人员进行定期考核。

- 建立合作制度。加强与相关单位的合作，及时获得必要的信息和技术支持。

除此之外，任何信息系统的运行都必须遵守国家的有关法律和规定，特别是关于计算机信息系统安全的法律规定。

## 2. 日常运行管理内容

信息系统的日常运行管理是为了保证系统能长期有效地正常运转，所以我们要记录系统的运行情况、系统运行的日常维护等工作情况。

对系统运行情况的记录应事先制定登记格式和登记要点。人工记录的系统运行情况和系统自动记录的运行信息，都应作为基本的系统文档按照规定的期限保管。这些文档既可以在系统出现问题时查清原因和责任，还能作为系统维护的依据和参考。

### (1) 系统运行情况的记录

原则上讲，从每天计算机的打开、应用系统的进入、功能项的选择与执行，到下班前的数据备份、存档、关机等，都要对系统软硬件及数据等的运作情况做记录。运行情况有正常、不正常与无法运行三种情况。可对正常情况不予记录，对于不正常情况和无法运行的情况则应将所见的现象、发生的时间及可能的原因做尽量详细的记录，因为这些信息对系统问题的分析与解决有重要的参考价值。

### (2) 审计踪迹

审计踪迹就是指系统中设置了自动记录功能，能通过自动记录的信息发现或判明系统的问题和原因。这里的审计有两个关键的地方：一是要每日都进行，二是对主要是技术要有一定的审查机制。在审计踪迹系统中，建立审计日志是一种基本的方法。通过日志，系统管理员可以了解到有哪些用户在什么时间、以什么样的身份登录到系统，也可以查到对特定文件和数据所进行的改动。

现在大多数的操作系统和数据库都提供了跟踪和自动记录功能，一些数据库系统中还提供审计踪迹数据字典，使用者可以用预先定义的审计踪迹数据字典视图来观察审计踪迹数据。对于审计内容，可以在三个层次上设定。

- 语句审计：语句审计是对于特定的数据库语句所进行的审计。例如，在一个系统文件中记录所有使用了 Create 命令的信息。
- 特权审计：对特定权限的使用所进行的审计。
- 对象审计：规定对特定的对象审计特定的语句。

### (3) 审查应急措施的落实

为了减少意外事件引起的对信息系统的损害，首先要制定应付突发性事件的应急计划，然后每日要审查应急措施的落实情况。应急计划主要针对一些突发性的、灾害性的事

件,例如火灾、水灾等。因此,机房值班员每日都应仔细审查相应器材和设备是否良好,相应的资源是否做好了备份等。

资源备份包括两个方面的工作,即数据备份和设备备份。数据备份是必须要做的,在关键的领域,还必须进行设备备份。应将备份文件复制到远离主机或文件中新的其他主机或者存储库中,保证备份文件存放在灾难事件影响不到的地方。

#### (4) 系统资源的管理

在维护信息系统正常运行的过程中还应对计算机的使用及打印机、墨粉的消耗等制定合理的管理办法。对不能充分满足用户需求的资源,一般可采用收费的方法来控制。收费既要能使系统的效能发挥到最大,使得信息系统部门的利益和管理措施得到保证,又要做到业务部门愿意接受,不能妨碍业务部门的正常使用。

### 3. 系统软件及文档管理

#### (1) 系统软件的管理除日常维护以外,还包括版本更新和升级等

由于计算机科学技术的迅速发展,新的软硬件不断推出,使系统的外部环境发生变化。这里的外部环境不仅包括计算机硬件、软件的配置,还包括数据库、数据存储方式在内的数据环境。为了适应企业的发展,版本更新和升级是必不可少的。

#### (2) 信息系统文档的管理

信息系统的文档与其他类型的文档一样,也具有它自身的生命周期,分为创建期、处理期、存储期、使用期、销毁期。每种文档都处于生命周期中的某一时期。当然周期的划分也不是绝对的,各周期有时是不能截然分开的。信息系统文档的生命周期普遍要比信息系统的生命周期长。也就是说,绝大多数信息系统的文档要在相应的信息系统淘汰3年~5年后才能销毁。

为了最终得到高质量的信息系统文档,在信息系统的建设过程中就必须加强对文档的管理。文档的管理应从以下几个方面着手进行。

- 文档管理的制度化:必须形成一整套的文档管理制度,其内容可以包含文档的标准、修改文档和出版文档的条件约束、开发人员在系统建设不同时期就其文档建立工作应承担的责任和任务,以及根据这一套完善的制度来最终协调、控制系统开发工作,并依次对每一个开发成员的工作进行评价等。
- 文档要标准化、规范化:在系统开发前必须首先选择或制定文档标准。在统一标准的前提下,开发人员负责建立所承担任务的文档资料。对于已有参考格式和内容的文档,应尽量按相应的规范撰写文档。而对于那些没有参考格式的文档,如需求变更申请书等,应该在项目组内部出台相应的规范和格式。
- 文档管理人员:项目小组应设文档组或至少一位文档保管人员,负责集中保管本

项目已有文档的两套主文本。两套文本内容应完全一致，其中一套可按一定手续办理借阅。

- 维护文档的一致性：信息系统开发建设过程是一个不断变化的动态过程，一旦对某一文档进行修改时，要及时、准确地修改与之相关联的文档，否则将会引起系统开发工作的混乱。而这一过程又必须有相应的制度来保证。
- 维持文档的可追踪性：由于信息系统开发的动态性，系统的某种修改是否最终有效，要经过一段时间的检验，所以文档要分版本来实现，而各版本的出版时机及要求也要有相应的制度来保障和约束。

信息系统的文档一般是按文件级来进行管理的，即以文件作为管理的基本单位。反映一份文件各个方面属性的集合就构成了一条记录，它一般应包括文档名、责任者、事件、密级、保管期限、分类号、关键词等项目。

## 5.5.2 系统维护知识

### 1. 系统可维护性概念

系统的可维护性可以被定义为：维护人员理解并修改这个软件的难易程度。提高系统的可维护性是开发管理信息系统所有步骤的关键目的所在。系统是否能被很好的维护，可以用系统的可维护性这一指标来衡量。

#### (1) 系统可维护性的评价指标

- 可理解性：指别人能理解系统的结构、界面功能和内部过程的难易程度。模块化设计、详细设计文档、结构化设计和良好的高级程序设计语言等，都有助于提高可理解性。
- 可测试性：诊断和测试的容易程度取决于易理解的程度。好的文档资料有利于诊断和测试，同时，程序的结构、高性能的测试工具以及周密的测试工序也是至关重要的。为此，开发人员在系统设计和编程阶段就应尽力把程序设计成易于诊断和测试的。此外，在系统维护时，应该充分利用在系统测试阶段保存下来的测试用例。
- 可修改性：诊断和测试的容易程度与系统设计所制定的设计原则有直接关系。模块的耦合、内聚、作用范围与控制范围的关系等都对可修改性有影响。

#### (2) 维护与软件文档

文档是软件可维护程度高低的决定因素。由于长期使用的大型软件系统在使用过程中必然经受多次修改，所以文档就显得非常的重要。

软件系统的文档可以分为用户文档和系统文档两类：用户文档主要描述系统功能和使用方法，而并不关心这些功能是怎样实现的；系统文档则主要描述系统设计、实现和测试

等各方面的内容。

可维护性是所有软件都应具有的基本特点，必须在开发阶段保证软件具有可维护的特点。在软件工程的每一个阶段都应考虑并提高软件的可维护性，在每个阶段结束前的技术审查和管理复查中，应该着重对可维护性进行复审。

在系统分析阶段的复审过程中，应该对将来要改进的部分和可能会修改的部分加以注解并指明，并且指出软件的可移植性问题以及可能影响软件维护的系统界面；在系统设计阶段的复审期间，应该从容易修改、模块化和功能独立的目的出发，评价软件的结构和过程；在系统实施阶段的复审期间，代码复审应该强调编码风格和内部说明文档这两个影响可维护性的因素。在完成了每项维护工作之后，都应该对软件维护本身进行认真的复审。

### (3) 软件文档的修改

维护应该针对整个软件配置，不应该只修改源程序代码。如果对源程序代码的修改没有反映在设计文档或用户手册中，可能会产生严重的后果。每当对数据、软件结构、模块过程或任何其他有关的软件特点有了改动时，必须立即修改相应的技术文档。我们要意识到，不能准确反映软件当前状态的设计文档可能比完全没有文档更坏。在以后的维护工作中很可能因文档不符合实际需求而导致不能正确的理解软件，从而在维护中引入过多的错误。

## 2. 系统维护的内容及类型

系统维护主要包括硬件设备的维护、应用软件的维护和数据的维护。

### (1) 硬件维护

硬件的维护应由专职的硬件维护人员来负责，主要有两种类型的维护活动：一种是定期的设备保养性维护，保养周期可以是一周或一个月不等，维护的主要内容是进行例行的设备检查与保养、易耗品的更换与安装等；另一种是突发性的故障维护，即当设备出现突发性故障时，由专职的维修人员或请厂方的技术人员来排除故障，这种维修活动所花时间不能过长，以免影响系统的正常运行。

### (2) 软件维护

软件维护主要是指根据需求变化或硬件环境的变化对应用程序进行部分或全部的修改。修改时应充分利用源程序，修改后要填写程序修改登记表，并在程序变更通知书上写明新老程序的不同之处。

软件维护的内容一般有以下几个方面。

- 正确性维护：是指改正在系统开发阶段已发生而系统测试阶段尚未发现的错误。这方面的维护工作量要占整个维护工作量的 17%~21%。所发现的错误有的不太重要，不影响系统的正常运行，其维护工作可随时进行；而有的错误非常重要，甚至影响整个系统的正常运行，其维护工作必须制定计划，进行修改，并且要进行复查和控制。



- **适应性维护**：是指使应用软件适应信息技术变化和管理需求变化而进行的修改。这方面的维护工作量占整个维护工作量的 18%~25%。由于目前计算机硬件价格的不断下降，各类系统软件层出不穷，人们常常为改善系统硬件环境和运行环境而提出系统更新换代的需求；企业的外部市场环境和管理需求的不断变化也使得各级管理人员不断提出新的信息需求。这些因素都将导致适应性维护工作的产生。进行这方面的维护工作也要像系统开发一样，需要有计划、有步骤地进行。
- **完善性维护**：这是为扩充功能和改善性能而进行的修改，主要是指对已有的软件系统增加一些在系统分析和设计阶段中没有规定的功能与性能特征。这些功能对完善系统功能是非常必要的。另外，还包括对处理效率和编写程序的改进，这方面的维护占整个维护工作的 50%~60%，比重较大，也是关系到系统开发质量的重要方面。这方面的维护除了要有计划、有步骤地完成外，还要注意将相关的文档资料加入到前面相应的文档中去。
- **预防性维护**：为了改进应用软件的可靠性和可维护性，为了适应未来的软硬件环境的变化，应主动增加预防性的新功能，以使应用系统适应各类变化而不被淘汰。比如将专用报表功能改成通用报表生成功能，以适应将来报表格式的变化。这方面的维护工作量占整个维护工作量的 4%左右。

### (3) 数据维护

数据维护工作主要是由数据库管理员来负责，主要负责数据库的安全性和完整性以及进行并行性控制。数据库管理员还要负责维护数据库中的数据，当数据库中的数据类型、长度等发生变化时，或者需要添加某个数据项时，要负责修改相关的数据库、数据字典，并通知有关人员。另外，数据库管理员还要负责定期出版数据字典文件及一些其他数据管理文件，以保留系统运行和修改的轨迹。当系统出现硬件故障并得到排除后要负责数据库的恢复工作。

数据维护中还有一项很重要的内容，那就是代码维护。不过代码维护发生的频率相对较小。代码的维护应由代码管理小组进行。变更代码应经过详细讨论，确定之后要用书面形式贯彻。代码维护的困难往往不在于代码本身的变更，而在于新代码的贯彻。为此，除了成立专门的代码管理小组外，各业务部门要指定专人进行代码管理，通过他们贯彻使用新代码。这样做的目的是要明确管理职责，有助于防止和更正错误。

### 3. 系统维护的管理和步骤

要强调的是，系统的修改往往会“牵一发而动全身”。程序、文件、代码的局部修改都可能影响系统的其他部分。因此，系统的维护工作应有计划有步骤的统筹安排，按照维护任务的工作范围、严重程度等诸多因素确定优先顺序，制定出合理的维护计划，然后通过一定的批准手续实施对系统的修改和维护。

通常对系统的维护应执行以下步骤:

(1) 提出维护或修改要求。操作人员或业务领导用书面形式向负责系统维护工作的主管人员提出对某项工作的修改要求。这种修改要求一般不能直接向程序员提出。

(2) 领导审查并做出答复,如同意修改则列入维护计划。系统主管人员进行一定的调查后,根据系统的情况和工作人员的情况,考虑这种修改是否必要、是否可行,做出是否修改、何时修改的答复。如果需要修改,则根据优先程度的不同列入系统维护计划。计划的内容应包括维护工作的范围、所需资源、确认的需求、维护费用、维护进度安排以及验收标准等。

(3) 领导分配任务,维护人员执行修改。系统主管人员按照计划向有关的维护人员下达任务,说明修改的内容、要求、期限。维护人员在仔细了解原系统的设计和开发思路的情况下对系统进行修改。

(4) 验收维护成果并登记修改信息。系统主管人员组织技术人员对修改部分进行测试和验收。验收通过后,将修改的部分嵌入系统,取代旧的部分。维护人员登记所做的修改,更新相关的文档,并将新系统作为新的版本通报用户和操作人员,指明新的功能和修改的地方。在进行系统维护过程中,还要注意维护的副作用。维护的副作用包括两个方面:一是修改程序代码有时会发生灾难性的错误,造成原来运行比较正常的系统变得不能正常运行,为了避免这类错误,要在修改工作完成后进行测试,直至确认和复查无错为止;二是修改数据库中数据的副作用,当一些数据库中的数据发生变化时,可能导致某些应用软件不再适应这些已经变化了的数据而产生错误。为了避免这类错误,一是要有严格的数据描述文件,即数据字典系统;二是要严格记录这些修改并进行修改后的测试工作。

总之,系统维护工作是信息系统运行阶段的重要工作内容,必须予以充分的重视。维护工作做得越好,信息系统的作用才能够得以充分发挥,信息系统的寿命也就越长。

### 5.5.3 系统评价知识

#### 1. 系统评价的目的和任务

信息系统的评价分为广义和狭义两种:广义的信息系统评价是指从系统开发的开始到结束的每一阶段都需要进行评价;狭义的信息系统评价则是指在系统建成并投入运行之后所进行的全面、综合的评价。

按评价的时间与信息系统所处阶段的关系,又可从总体上把广义的信息系统评价分成立项评价、中期评价和结项评价。

- 立项评价:指信息系统方案在系统开发前的预评价,即系统规划阶段中的可行性研究。评价的目的是决定是否立项进行开发,评价的内容是分析当前开发新系统的条件是否具备,明确新系统目标实现的重要性和可能性,主要包括技术上的可

行性、经济上的可行性、管理上的可行性和开发环境的可行性等方面。由于事前评价所用的参数大都是不确定的，所以评价的结论具有一定的风险性。

- 中期评价：项目中期评价包含两种含义，一是指项目方案在实施过程中，因外部环境出现重大变化，比如市场需求变化、竞争性技术或更完美的替代系统的出现，或者发现原先设计有重大失误等，需要对项目的方案进行重新评估，以决定是继续执行还是中止该方案；另一种含义也可称为阶段评估，是指在信息系统开发正常情况下，对系统设计、系统分析、系统实施阶段的阶段性成果进行评估，由于一般都把阶段性成果的提交视为信息系统建设的里程碑，所以，阶段评估又可叫里程碑式评价。
- 结项评价：信息系统的建设是一个项目，是项目就需要有终结时间。结项评价是指项目准备结束时对系统的评价，一般是指在信息系统投入正式运行以后，为了了解系统是否达到预期的目的和要求而对系统运行的实际效果进行的综合评价。因此，结项评价又是狭义的信息系统评价。信息系统项目的鉴定是结项评价的一种正规的形式。结项评价的主要内容包括系统性能评价、系统的经济效益评价以及企业管理效率提高、管理水平改善、管理人员劳动强度减轻等间接效果。通过结项评价，用户可以了解系统的质量和效果，检查系统是否符合预期的目的和要求；开发人员可以总结开发工作的经验、教训，这对今后的工作十分有益。

在对信息系统进行评价考核的时候，应该注意以下几个问题。

首先，信息通过基本资料录入、进货、订货、盘点、零售等各个环节采集进来，其中任意一个环节的数据录入出现问题，都将导致最终报表的不准确，而报表不准确就意味着企业决策者无法根据报表决定企业的运作，更谈不上数据分析和决策支持了。这也是我们目前大部分使用了信息系统的企业普遍存在的问题。究竟是什么原因导致了数据采集的不准确呢？一些企业错误地将数据准确性作为考核信息部的一个指标，其实数据不准确更多源于管理上存在的问题，正因为数据源头非常多，所造成的数据不一致的问题不是信息部都能解决的，最终数据采集的成败将由最高管理层对数据采集各环节的管理力度所决定，而数据采集的成败又将最终决定整个信息系统应用的成败。

此外，信息系统并不是万能系统。系统应用的过程中，有些问题是信息系统擅长解决的，如大量的、重复的、规范性的事务处理；而有些问题是信息系统不擅长解决的，如特殊的、偶然的、不规范的经营管理内容。让信息系统做不擅长的工作，势必在应用的过程中，导致投入的管理成本远远大于它所产生的效益。对这种灵活、多变的情况，不妨采用人工处理或通过制度的限制，尽量避免不规范的行为频繁发生，从而真正实现企业简单复制、快速扩张、规模效益的目的。

## 2. 系统评价的指标

我们从以下几方面综合考虑，建立起一套指标体系理论框架。

- 从信息系统的组成部分出发，信息系统是一个由人机共同组成的系统，所以可以按照运行效果和用户需求（人）、系统质量和技术条件（机）这两条线索构造指标。
- 从信息系统的评价对象出发，对于开发方来说，他们所关心的是系统质量和技术水平；对于用户方而言，关心的是用户需求和运行质量；系统外部环境则主要通过社会效益指标来反映。
- 从经济学角度出发，分别按系统成本、系统效益和财务指标等三条线索建立指标。

## 第 6 章 嵌入式系统设计

### 6.1 嵌入式系统设计的特点

嵌入式系统设计的主要任务是定义系统的功能、决定系统的架构，并将功能映射到架构。这里的架构既包括软件系统架构也包括硬件系统架构。嵌入式系统的设计方法跟一般的硬件设计、软件开发的方法不同，是采用硬件和软件协同设计的方法，开发过程不仅涉及软件领域的知识，还涉及硬件领域的综合知识，甚至还涉及机械等方面的知识。要求设计者必须熟悉并能自如地运用这些领域的各种技术，才能使设计的系统达到最优。

与通常的系统设计相比，嵌入式系统的设计具有以下几个特点：

#### (1) 软/硬件协同并行开发。

软/硬件协同并行开发就是在整个设计的生命周期，软件和硬件的设计一直是保持并行的，在设计过程中两者交织在一起，互相支持，互相提供开发的平台，而不是传统方法中将软/硬件的设计分开独立进行，在设计流程的开始就将系统所要实现的功能划分到用硬件或软件实现，然后独立进行硬件和软件设计，最后才进行软硬件的集成。系统是否满足用户需求只有等到软硬件集成之后进行测试才能知道，所以传统设计方法进行复杂系统的设计时，常常难以达到设计要求和实现优化设计。

#### (2) 嵌入式系统通常是面向特定应用的系统。

嵌入式 CPU 与通用型的 CPU 最大不同就是，嵌入式 CPU 大多工作在为特定用户群设计的系统中，它通常都具有低功耗、小体积、高集成度等特点，能够把通用 CPU 中许多由板卡完成的任务集成在芯片内部，从而有利于嵌入式系统设计趋于小型化。因此，元件的移动能力大大增强，同时跟网络的耦合也越来越紧密。

#### (3) 实时嵌入式操作系统的多样性 (RTOS)。

实时操作系统不像台式机操作系统那样，只有微软公司一家独大。现在可用的实时操作系统很多。如 VxWorks, QNX, uc/os, RT-Linux, WinCE, Psos 等。可以根据自己的需求，选择相应的操作系统。

#### (4) 与台式机相比，可利用系统资源很少。

#### (5) 嵌入式系统设计需要交叉开发环境。

嵌入式系统的开发通常采用“宿主机/目标机”方式（如图 6-1 所示）。首先，利用宿

主机上丰富的设备资源以及良好的开发环境来开发和仿真调试目标机上的软件。然后，通过 UART 接口或 Ethernet 接口将交叉编译生成的目标代码传输并下载到目标机上，并用交叉调试器在实时内核/操作系统或监控程序的支持下进行分析与调试。最后，目标机在特定的环境下运行。

宿主机 (Host) 是一台通用的计算机，一般是 PC 机。它通过串行接口或网络连接与目标机进行通信。宿主机上的软/硬件资源比较丰富，包括功能强大的操作系统，如 Windows 和 Linux，以及各种各样的开发调试工具，如 GNU 的嵌入式开发工具套件等。这些辅助的开发工具能够极大的提高系统的开发效率。

目标机 (Target) 常用在嵌入式系统的开发过程期间。目标机可以是嵌入式系统的实际运行环境，也可以是能替代实际环境的仿真系统。通常目标机的体积较小，集成度高，外围设备较多，输入设备有键盘、触摸屏等；输出设备有 LCD、LED 等。因为目标机的资源有限，所以在目标机上运行的软件通常需要根据实际具体的要求进行裁减和配置。目标机的应用软件常与实时操作系统绑定一起运行。

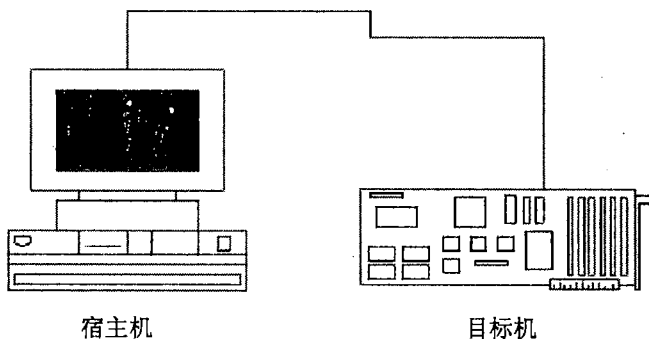


图 6-1 宿主机/目标机开发方式

#### (6) 嵌入式系统的程序需要固化。

通用的系统在测试完成后就可以直接投入使用，其目标环境一般是 PC 机，因此在总体结构上与开发环境差别不大，而嵌入式系统的开发环境是 PC 机，但运行的目标环境基本是千差万别，可以是手机、PDA，也可以是仪器设备等，而且应用软件在目标环境下必须存储在非易失性存储器中（如 Flash、ROM 等），保证用户关机之后下次能够再次使用。因此，在系统应用软件开发完成之后，应生成固化版本，将其烧写到目标环境的 Flash Memory 中运行。

#### (7) 嵌入式系统的软件开发难度较大。

嵌入式系统的特点之一就是系统要求具有实时性。这些实时性在开发的系统中要得到

保证,就要求设计者在系统的需求分析中充分考虑系统的实时性。实时性的体现一部分来源于实时操作系统的实时性,这方面可以采用实时操作系统,如 RT-Linux、VxWorks、Hopen OS 等,而另一部分依赖于系统本身的设计和代码的质量,这就要求系统的设计者和开发人员构建出良好的系统模型和算法,所有的这些必然会增加软件的开发难度。

(8) 嵌入式应用软件的开发需要强大的开发工具和操作系统的支持。

随着 Internet 技术的成熟、带宽的提高,ICP 和 ASP 在网上提供的信息内容日趋丰富、应用项目多种多样,像电话、手机、电话座机及电冰箱、微波炉等嵌入式电子设备的功能不再单一,电气结构也更为复杂。为了满足应用功能的升级,设计师们一方面采用更强大的嵌入式处理器,如 32 位、64 位 RISC 芯片或信号处理器 DSP 增强处理能力;同时还采用实时多任务编程技术和交叉开发工具技术来控制功能复杂性,简化应用程序设计、保障软件质量和缩短开发周期。

(9) 嵌入式系统还需要提供强大的硬件开发工具和软件包的支持,需要设计者从速度、功能和成本综合考虑。此外,嵌入式系统对稳定性、可靠性、功耗、抗干扰性、重量体积等方面的性能要求都比通用系统的要求更为严格。

## 6.2 嵌入式系统的设计流程

嵌入式系统的设计和开发必须将所有的硬件、软件、人力资源等集中起来,并且进行适当的组合,以实现目标系统对性能和功能等各方面的需求。在嵌入式系统的开发过程当中,实时性、可靠性、功耗等都与功能一样重要,这就使嵌入式系统开发关注的方面更广泛,要求的精度也更高。

嵌入式系统的设计和开发流程一般分为以下几个阶段:产品定义(即系统需求分析阶段、规格说明阶段)、硬件和软件划分、迭代与实现、详细的硬件与软件设计、硬件与软件集成、系统测试和系统维护与升级。各个阶段通常需要不断的反复和修改,直到完成最终设计目标。

嵌入式系统设计的流程如图 6-2 所示。

嵌入式系统的设计过程并不像图 6-2 所示那样简单,在各阶段内部及各阶段之间会发生大量的迭代与优化,前一步的设计缺陷会直接导致后面阶段设计任务无法完成,从而必须回到起点重新进行设计,这就会造成生产成本的提高,增加产品的开发时间,造成不必要的损失。因此,前期的工作一定要精细,确保准确无误,尽量减少因前期工作所造成的损失。下面对各阶段给出具体的描述。

### (1) 产品定义

产品定义相当于一般软件工程中的需求分析阶段。即对产品需求加以分析、细化,并

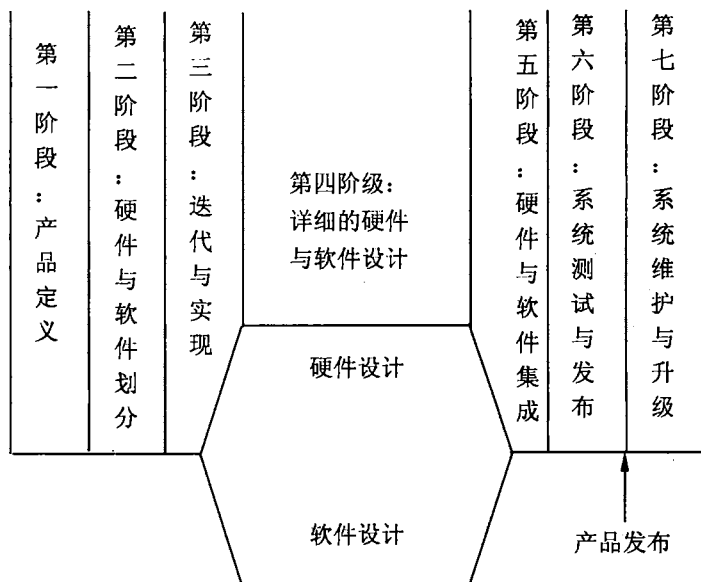


图 6-2 嵌入式系统的设计流程

抽象出需要完成的功能列表，明确定义所要完成的任务。

#### (2) 软件与硬件的划分

由于嵌入式设计分为硬件与软件设计两个部分，设计人员必须确定系统的哪些功能由硬件完成，那些功能由软件完成，这种选择成为“划分决策”，即软件与硬件的划分。

#### (3) 迭代与实现

迭代与实现阶段是软硬件划分阶段的延续，随着软硬件被初步划分，软硬件设计小组分别对软硬件进行建模，随着软硬件建模过程的深入，更多的设计约束被理解，此时可以修改软硬件划分的界限，实现对软硬件更为合理的划分。

#### (4) 详细的硬件与软件设计

随着上一个阶段的完成，系统被合理地划分成了软硬件两个部分，此阶段是对系统的软硬件分别进行实现的过程。在软硬件的实现过程中分别有自己的设计方法和技巧，这将在下几个部分加以介绍。

#### (5) 硬件和软件的集成

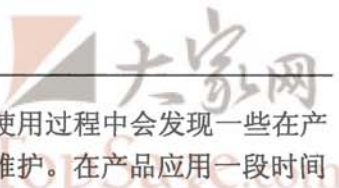
将已实现的硬件模块和软件功能模块综合、整合为统一的系统。

#### (6) 系统测试与发布

嵌入式系统一般具有严格的设计界限，以达到成本目标，所以测试必须查明系统是否在运行时能接近最优性能。而且嵌入式系统要求在运行时具有相当高的可靠性，因此在产品发布之前必须进行产品的严格测试。

#### (7) 系统维护与升级





在产品发布之后还要不断对产品进行维护和升级。产品在使用过程中会发现一些在产品的设计阶段没有想到的问题，对这些问题的解决就是产品的维护。在产品应用一段时间后，用户会对产品提出更多的需求，通过对产品的升级可以解决用户不断增加的需求。

### 6.2.1 产品定义

嵌入式系统一般是面向产品开发的，因此在系统开发初期需要的产品定义就是要搞清楚需要完成的任务。它相对于一般软件工程的需求分析阶段，即需要搞清楚该产品的功能及产品的性能，并将功能整理，确定设计任务和设计目标。产品功能相对于一般软件工程需求分析的功能性需求，它包括系统的基本功能，如输入输出、操作方式等；产品的性能相对于一般软件工程的非功能性需求，它包括系统性能、成本、功耗、体积、重量等因素。

产品定义是一项复杂而费时的的工作，它是整个系统设计的第一步，因此重要性是显而易见的。产品定义做的完整与否，将会决定最终受市场欢迎的情况。为了使产品定义更加清晰明确，可以使用一个产品定义表格（如表 6-1 所示），它将系统的功能和性能需求综合起来。

表 6-1 需求分析表格

项 目	描 述
名称	每一个工程项目都有自己的名称，它是所要完成项目的总体概括
目的	关于系统将要满足的需求的简单描述，概要介绍所设计系统的主要特征
输入和输出	系统的输入输出包括数据的类型（如模拟信号、数字信号等）、数据特性（如是否是用户的输入、数据的位数等）、输入输出设备的类型（如按键、显示器等）
功能	对系统所要做的工作的具体详细的描述。可以分析系统从输入到输出的流程，当系统接收到输入时，执行哪些动作
性能	系统所要求处理的速度、实时性和实用性，例如 0.25 秒内要求更新一次屏幕
生产成本	任何系统的设计都要考虑到成本问题，主要的是硬件构件和人员的花费，产品的价格最终会影响系统的体系结构，这就需要对最终产品的价格有一个粗略的估价
功耗	嵌入式系统的特点决定了有些设备是靠电池供电。靠电池供电的系统必须认真地对功耗问题进行考虑
物理尺寸和重量	物理尺寸和重量 最终产品的物理特性会因为使用的领域不同而大不相同。有些系统对重量没有什么约束，如一台控制装配线的工业控制系统通常装配在一个标准尺寸的柜子里。有些则不同，如手持设备对系统的尺寸和重量就有很严格的限制。对系统的物理尺寸和重量有一定的了解有助于对系统体系结构的设计

例如，便携式网络电视设计的需求分析如下。

名称：便携式网络电视

目的：为用户提供移动网络和收看数字电视服务，同时具有所有广播和交互式多媒体

应用功能。

输入：触摸式面板，一个电源按钮

输出：LCD 显示屏，内置喇叭

功能：

(1) 电子节目指南 (EPG)。给用户提供一个容易使用、界面友好、可以快速访问想看节目的一种方式，用户可以通过该功能看到一个或多个频道甚至所有频道近期将播放的电视节目。

(2) 高速数据广播。能给用户 provide 股市行情、票务信息、电子报纸、热门网站等各种消息。

(3) 软件在线升级。软件在线升级可看成是数据广播的应用之一。数据广播服务器按 DVB 数据广播标准将升级软件广播下来，便携式网络电视能识别该软件的版本号，在版本不同时接收该软件，并对保存在存储器中的软件进行更新。

(4) Internet 接入和电子邮件。便携式网络电视可通过内置的无线网卡方便地实现因特网接入功能。用户可以通过平台内置的浏览器上网，发送电子邮件。

(5) 有条件接收。有条件接收的核心是加扰和加密，便携式网络电视应具有解扰和解密功能。

性能：画面流畅清晰，刷新速率>30 帧/秒

## 6.2.2 嵌入式系统的软硬件划分

随着芯片设计和制造技术水平的发展，微处理器的运算速度得到很大提高，因此很多传统上必须由硬件实现的功能现在可以使用软件实现。与此同时，近年来，FPGA 技术的提高和大容量、低成本的新型 FPGA 的出现，为高性能的数字控制系统提供了新的实现方法。可以说，以嵌入式微处理器和 FPGA 为核心的系统设计技术代表了现代数控系统的软件和硬件实现方法。

但是，微处理器的运算资源和 FPGA 的逻辑资源还是有限的，而且微处理器擅长的是串行的数据处理，而 FPGA 擅长的是并行的逻辑处理。因此就出现了功能实现的软硬件划分的问题。

需要注意的是，这里所指的有软硬件划分需要的功能都是那些既可以用软件实现又可以用硬件实现的功能，具体到实际的物理系统中，就是那些既可以用微处理器系统实现也可以用 FPGA 或模拟元件实现的功能。

在软硬件划分的问题上，一般遵循以下几个原则：

(1) 性能原则

不管使用软件还是硬件实现特定的功能，首先要满足性能要求，这是最重要的。例如

对于所有的模拟功能，虽然有模拟 FPGA 出现，但是其技术还不成熟，而数字脉冲输出加滤波的方法在相应速度、精度上无法与模拟元件相比，因此显然是必须由模拟元件，也就是硬件来实现。

### (2) 性价比原则

大容量 FPGA 理论上和实际上都可以完成本系统所要实现的全部数字功能，但是使用昂贵的 FPGA 来实现普通的微处理器就可以实现的功能是一种巨大的浪费。同时，为了让本系统的某些功能用软件实现而采用超高速的微处理器也是一种浪费。因此，可以说对于本系统来说，最重要的是分析对于同一个功能，是用微处理器来实现所需要的成本低，还是用 FPGA 来实现所需要的成本低。

### (3) 资源利用率原则

新型的微处理器往往集成了大量的外围元件，如串行接口、计数器、PWM 和 AD 等，也就常说的片上系统 SOC (System On Chip)，在性能相同的情况下，价格却比分立系统低廉。因此，很多系统面临的情况是使用了 SOC 芯片以后，不但可以实现那些符合高性价比原则的功能，还会有一些剩余的资源，如微处理器的计算资源。FPGA 的使用特点决定了不能完全按照估计的实际逻辑资源使用量去选型，而是要选择逻辑容量比实际可能需要的最大容量还要大 30% 的型号。而同一个系列的 FPGA 里虽然有内部逻辑资源容量不同的多种型号，但是找到完全符合选型要求的元件的机会还是比较小的，因此就必须选择刚好大于选型需求的元件。如此一来，不管是微处理器还是 FPGA 的资源都会有一些剩余。很多使用软件和硬件都可以满足性能要求的功能就必须考虑实现的方法，如键盘扫描功能。对于这些功能的实现方法就需要考虑软件和硬件的利用率原则，即不要把一方的资源用光，尽量使两方的资源利用率相等。由于微处理器所运行的控制软件的性能与微处理器的使用率有关，当使用率越低时，软件的相应速度会越快，或者可以以较低速度来运行微处理器。而且 FPGA 的运行速度与逻辑资源也有关系，逻辑资源利用率越低，可以运行的速度越快。而数字系统的运行速度与可靠性是有直接关系的。当运行速度接近极限速度时，可靠性就会降低，因此适当保留一定的提速空间对与提高系统的可靠性是有很大帮助的，同样也有利于将来的性能升级和维护。

3 个原则之间不是相互独立的，而是互相影响的，当具体实施时要同时考虑这几个原则，从而做出最优的选择。

## 6.2.3 嵌入式系统硬件设计

规格说明只是说明系统做些什么，具有哪些方面的功能，而不讲系统如何去做，怎样具体地实现。描述系统如何实现那些功能是体系结构所要做的。体系结构是系统整体结构的一个规划和描述，设计完成之后用于构建整个体系结构的构件。

硬件是嵌入式系统运行的载体也是嵌入式系统的基础，嵌入式系统硬件为嵌入式系统软件提供了执行环境，限定了嵌入式系统软件能够访问的资源。嵌入式系统所能完成的功能首先从硬件上得以体现。嵌入式系统的硬件设计是在嵌入式系统软硬件划分的基础上，对划分为硬件部分的功能单元所进行的设计。通常，嵌入式系统的硬件设计包含以下几个部分。

### 1. 嵌入式系统硬件的选择

嵌入式系统硬件的选择包括硬件平台和嵌入式处理器的选择、外围设备的选择和接口电路的选择。

#### 1) 硬件平台的选择

(1) 如果已有的系统实现了相似的功能，重用该结构是个很好的选择。

(2) 如果这是个全新的项目，考虑这些功能是否能用一个处理器实现。单个处理器是最容易实现和调试的。

(3) 如果该应用需要用多个处理器，最好选用能够满足需求的最少处理器实现。

(4) 在多处理器设计中，把控制和管理用一个处理器实现，这样简化了操作。系统中其他处理器处理系统中的工作负载。

(5) 按照这种方式设计的多处理器系统，可以从一个小的系统扩充为一个大的系统，通过增加处理器实现。因此客户可以从简单的处理器入手，随着系统负载的增加而增加。

#### 2) 处理器的选择

嵌入式系统的核心部件是嵌入式微处理器。设计者在选择处理器时要考虑的主要因素有以下几方面。

##### (1) 处理器的性能

处理器的性能取决于多个方面的因素，如处理器的时钟频率，内部寄存器的大小，指令字的长度等。对于许多需要使用处理器的嵌入式系统设计来说，目标不是在于挑选速度最快的处理器，而是在于选取能够满足系统个方面要求的处理器。

##### (2) 处理器的技术指标

目前，许多嵌入式处理器都集成了外围设备的功能，减少了芯片的数量，增强了系统的功能，降低了整个系统的开发费用。设计人员首先考虑的是，系统所要求的一些硬件能否无需过多的胶合逻辑（Glue Logic, GL）就可以连接到处理器上。其次是考虑该处理器的一些支持芯片，如 DMA 控制器、内存管理器、中断控制器、串行设备和时钟等的配套。

##### (3) 功耗

嵌入式微处理器最大并且增长最快的市场是手持设备、电子记事本、PDA、手机、GPS 导航器、智能家电等消费类电子产品。这些产品中选购的微处理器，典型的特点是要求高

性能、低功耗。这也是嵌入式系统重要的特点，目前这个领域已经吸引了许多 CPU 的生产厂商。今天，用户可以买到一个嵌入式的微处理器，其速度像笔记本中的 Pentium 一样快；而它仅使用普通电池供电即可，并且价格很便宜。如果用于工业控制，则对这方面的考虑较弱。

#### (4) 软件支持工具

仅有一个好的处理器，没有较好的软件开发工具的支持也是不行的，因此选择合适的软件开发工具对系统的实现会起到很重要的作用。

#### (5) 处理器是否内置调试工具

处理器如果内置调试工具可以大大缩小调试周期，降低调试的难度。

#### (6) 供应商是否提供评估板

许多处理器供应商可以提供评估板来验证理论是否正确，决策是否得当。

此外，硬件部件选择的其他因素还有生产规模、开发市场的目标、软件对硬件的依赖性。

### 2. 硬件功能模块划分

在嵌入式系统硬件选择完成之后，就要进行系统硬件功能模块的划分，这主要是对系统硬件资源进行合理的布局。硬件布局是针对于不同的硬件模块、硬件模块与嵌入式处理器之间的连接关系和模块之间的连接关系对硬件位置所做的调整。

### 3. 硬件的可靠性

系统的相当一部分可靠性体现在硬件设计的可靠性上，如果硬件不能可靠稳定的运行，那么嵌入式系统就不可能稳定的运行。可以人为地加入硬件保护电路，硬件检测电路，或硬件复位电路来保证硬件可靠的运行。

## 6.2.4 嵌入式系统的软件设计

嵌入式系统的软件设计不同于 Windows 环境下的软件编程，在嵌入式开发过程中有主机和目标机的角色之分。主机是执行编译、链接、定址过程的计算机；目标机是指运行嵌入式软件的硬件平台。首先须把应用程序转换成可以在目标机上运行的二进制代码。这一过程包含 3 个步骤：编译、链接、定址。编译过程由交叉编译器实现。所谓交叉编译器就是运行在一个计算机平台上并为另一个平台产生代码的编译器。编译过程产生的所有目标文件被链接成一个目标文件，称为链接过程。定址过程会把物理存储器地址指定给目标文件的每个相对偏移处。该过程生成的文件就是可以在嵌入式平台上执行的二进制文件。

在软硬件划分阶段之后，就进入到了软硬件分别实现阶段。嵌入式软件是实现嵌入式系统功能的关键，根据嵌入式软件的特点和实际当中嵌入式系统的具体要求，在进行嵌入式系统软件设计时。通常要考虑以下几个方面因素或达到其提出的要求。

## 1. 嵌入式软件平台的选择

### 1) 平台选择

(1) 平台的特殊需求（该平台是否需要实时操作系统的支持）。

(2) 对该硬件和软件平台的使用情况。

(3) 支持该硬件和软件平台公司的财务状况如何，当然不能选择不能提供硬件/软件平台支持的公司的产品。

(4) 提供该平台公司的发展目标是什么，当然不希望选择的平台没有一个清晰的升级途径。

(5) 该平台是否有合适的开发工具。

(6) 可以使用该平台开发的开发人员有多少，对开发人员培训的费用是多少。

(7) 预留性能。

(8) 该平台需要完善吗？一个好的平台比一个需要完善的平台要便宜得多。

(9) 平台的设备驱动程序（是否能够提供用户开发的设备驱动程序）。

(10) 平台支持哪些通信协议，如平台是否支持 TCP/IP、HTTP、UDP 等。

### 2) 操作系统的选择

硬件方案确定完成之后，操作系统的选择就相对容易一些了。硬件的不同，会影响操作系统的选择。通常操作系统的选择需要考虑到以下几个方面：

(1) 操作系统本身所提供的开发工具。有些实时操作系统（RTOS）只支持该系统供应商的开发工具，因此，还必须向操作系统供应商获取编译器和调试器等；而有些操作系统使用广泛，且有第三方工具可用，因此，选择的余地比较大。

(2) 操作系统向硬件接口移植的难度。操作系统到硬件的移植是一个重要的问题，是关系到整个系统能否按期完工的一个关键因素。因此，要选择那些可移植性程度高的操作系统，避免操作系统难以向硬件移植而带来的种种困难，加快整个系统的开发进度。

(3) 操作系统的内存要求。均衡考虑是否需要额外花钱去购买 RAM 或 EEPROM 来迎合操作系统对内存的较大要求。

(4) 开发人员是否熟悉此操作系统及其提供的系统 API。

(5) 操作系统是否提供硬件的驱动程序，如网卡驱动程序等。

(6) 操作系统的是否具有可剪裁性。有些操作系统具有较强的可剪裁性，如嵌入式 Linux、Tornado/VxWorks 等等。

(7) 操作系统的实时性能。

### 3) 编程语言的选择

编程语言的选择主要考虑以下因素：

(1) 通用性。不同种类的微处理器都有自己专用的汇编语言。这就为系统设计者设置





了一个巨大的障碍，使系统编程更加困难，软件重用无法实现。而高级语言一般和具体机器的硬件结构联系较少，多数微处理器都有良好的支持，通用性较好。

(2) 可移植性程度。汇编语言和具体的微处理器密切相关，为某个微处理器设计的程序不能直接移植到另一个不同种类的微处理器上使用，移植性差；而高级语言对所有微处理器都是通用的，程序可以在不同的微处理器上运行，可移植性较好。

(3) 执行效率。一般来说，越是高级的语言，其编译器和开销就越大，应用程序也就越大、越慢；但单纯依靠低级语言，如汇编语言来进行应用程序的开发，带来的问题是编程复杂、开发周期长。因此，存在一个开发时间和运行性能间的权衡问题。

(4) 可维护性。低级语言如汇编语言，可维护性不高。高级语言程序往往是模块化设计，各个模块之间的接口是固定的。当系统出现问题时，可以很快地将问题定位到某个模块内，并尽快得到解决。另外，模块化设计也便于系统功能的扩充和升级。

#### 4) 集成开发环境考虑的因素

集成开发环境 (Integrated Development Environment, IDE) 应考虑以下因素：

(1) 系统调试器的功能。系统调试特别是远程调试是一个重要的功能。

(2) 支持库函数。许多开发系统提供大量使用的库函数和模板代码，例如大家比较熟悉的 C++ 编译器就带有标准的模板库。它提供了一套用于定义各种有用的集装、存储、搜寻、排序对象。与选择硬件和操作系统的原则一样：除非必要，尽量采用标准的 glibc。

(3) 编译器开发商是否持续升级编译器。

(4) 连接程序是否支持所有的文件格式和符号格式。

#### 5) 硬件调试工具的选择

好的软件调试程序可以有效地发现大多数的错误，但是如果再选择一个好的硬件调试就会达到事半功倍的效果。常用的硬件调试工具有以下几种：

(1) 实时在线仿真器 (In-Circuit Emulator, ICE)。用户从仿真插头向 ICE 看，ICE 应是一个可被控制的 MCU。ICE 是通过一根短电缆连接到目标系统上的。该电缆的一端有一个插件，插到处理器的插座上，而处理器则插到这个插件上。ICE 支持常规的调试操作，如单步运行、断点、反汇编、内存检查、源程序级的调试等。

(2) 驻留监控软件。驻留监控程序运行在目标板上，PC 机端调试软件可以通过并口、串行接口、网口与之交互，以完成程序执行、存储器及寄存器读写、断点设置等任务。

(3) ROM 仿真器。ROM 仿真器用于插入目标上的 ROM 插座中的元件，用于仿真 ROM 芯片。可以将程序下载到 ROM 仿真器中，然后调试目标上的程序，就好像程序烧结在 PROM 中一样，从而避免了每次修改程序后直接烧结的麻烦。

(4) JTAG 仿真器。通过 ARM 芯片的 JTAG 边界扫描口与 ARM 核进行通信，不占用

目标板的资源，是目前使用最广泛的调试手段。

#### 6) 软件组件的选择

有些软件组件是免费的，有些软件组件是授权的。授权软件组件的费用一般都很高，但大都经过严格的测试，可靠性高，调试时间短。现在也有一些免费的自由软件组件，它们的性能、可靠性也很好。因此开发人员在选择的时候要加以权衡，确定哪种方案更好。

### 2. 软件的实时性设计

嵌入式应用一般都有实时性要求，尤其是在控制领域的应用中，即系统对于一个激励要在规定的时间内做出反应，否则将会对控制系统造成严重的后果。因此，在嵌入式系统软件设计时，要充分考虑实时性要求，对任务按照实时性分类，分别进行设计。

### 3. 软件的可靠性设计

嵌入式应用也有可靠性的要求，除硬件必须具备的可靠性之外，软件的可靠性也必不可少。在软件设计中要充分考虑软件运行条件，减少程序错误执行的可能，在程序运行错误的情况下能够进行排错处理，并重新恢复软件的正确执行。

### 4. 软件的可扩展设计

软件的可扩展设计是指在系统升级或者软件进一步开发时，新编写的程序能够很容易地插入到原来的程序中，可以不必改写（或较少改写）原来的代码。这样有利于系统功能的扩展和升级。

### 5. 嵌入式软件开发流程

嵌入式应用软件的开发必须将硬件、软件、人力资源等元素集成起来，并进行适当的组合以实现目标应用对功能和性能的需求。在嵌入式开发中，实时性能常常与功能一样重要，这就使嵌入式软件开发关注的方面更广泛，要求的精度更高。

如图 6-3 所示，嵌入式应用软件的开发流程与通用软件的开发流程大体相同，但在开发所使用的设计方法上有一定的差异。整个软件的开发流程可分为在软硬件划分阶段确定硬件驱动接口阶段、软件功能模块按照实时性进行划分阶段、各软件功能模块的代码生成阶段、软件功能模块的集成测试阶段、代码固化及固化后的调试阶段。

#### (1) 在软硬件划分阶段确定硬件驱动接口阶段

在软硬件划分阶段就要开始确定硬件驱动层，确定硬件驱动的软件接口，所有嵌入式系统软件都要在这个硬件驱动层基础之上来实现，因此可以说从软硬件划分确定了硬件驱动接口时，就开始了软件设计。

#### (2) 软件功能模块按照实时性进行划分阶段

此阶段是将系统软件划分为实时部分和分时部分。由于实时和分时部分的要求不同，



因此这样划分可以按照不同要求，分别对实时部分和分时部分进行实现，在满足要求的情况下，简化了整个系统的实时性设计。

### (3) 各软件功能模块的代码生成阶段

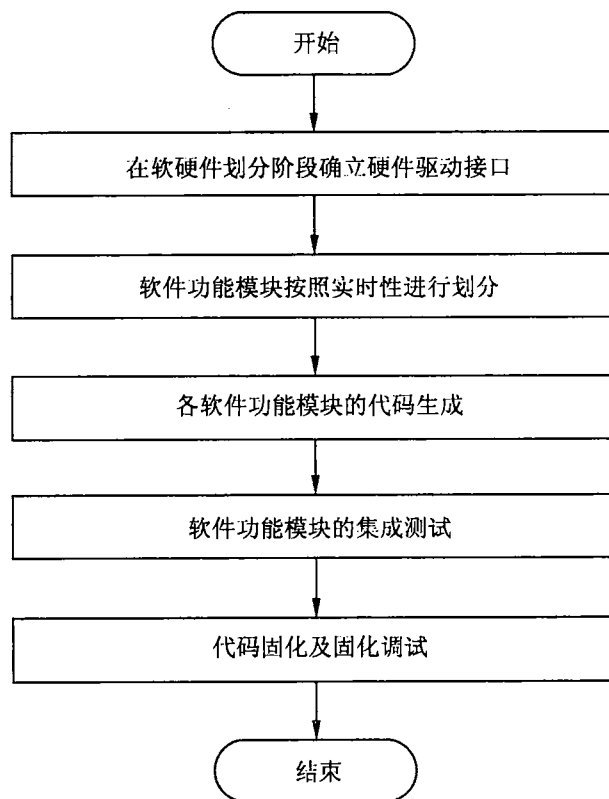


图 6-3 嵌入式软件设计流程

此阶段将对系统各个功能模块分别进行实现，是具体的代码编写和生成阶段。

### (4) 软件功能模块的集成测试阶段

在系统各功能模块编写完成后，要对功能模块进行集成测试，发现功能模块设计时的  
问题，以及各模块间的协调问题。

### (5) 代码固化以及固化后的调试阶段

在以上所有阶段完成后，就进入了代码的固化及固化后的调试阶段。此时将对系统代码移植到目标机上进行固化，脱离调试环境进行运行。

### 6.2.5 系统集成和测试

在系统的硬件构件和软件构件建立起来后,将硬件构件、软件构件和执行装置集成在一起才能得到一个可以运行的系统。在系统的集成过程当中,不能只是简单地把所有的东西插在一起,通常系统集成时会发现一些错误和问题,为了能够快速找到这些错误并能够准确定位到错误的位置,可以分阶段架构整个系统并且正确运行事先选择好的测试程序。如果每次只是对其中的一部分模块进行查错和纠错,那么就会很容易发现和识别其中简单的错误。只有在早期及时改正这些简单的错误,才能在以后的系统集成过程当中发现那些比较复杂或是难找到的严重错误,从而降低了负担,提高了整个系统开发的效率,缩短了开发的周期。因此,必须确保在体系结构和各个构件的设计阶段尽可能地按阶段组装系统和相对独立地测试各个模块的功能,看其是否满足规格说明书中给定的功能要求。

嵌入式系统集成过程中使用的调试工具很有限,它比桌面系统中可用的工具少得多,通常只有几种(常用的调试工具在前面已经简单介绍过),应根据实际的开发要求和实际的条件进行选择。

嵌入式系统的软件测试与通用软件的测试相似,分为单元测试和系统的集成测试。常用有黑盒测试和白盒测试两种测试方法。黑盒测试法把程序看成一个黑盒子,完全不考虑程序的内部结构和处理过程。黑盒测试是在程序接口进行的测试,它只检查程序功能是否能按照规格说明书的规定正常使用,程序是否能适当地接收输入数据产生正确的输出信息,并且保持外部信息的完整,黑盒测试又称为功能测试。白盒测试法的前提是可以把程序看成装在一个透明的白盒子里,也就是完全了解程序的结构和处理过程。这种方法按照程序内部的逻辑测试程序,检验程序中的每条通路是否都能按预定要求正确工作,白盒测试又称为结构测试。黑盒测试发现程序中所有错误的可能性不大,但它和白盒测试结合起来使用时,会产生一个非常好的测试集,因为黑盒能找到那些从代码结构中抽象出来的测试方法不能发现的错误。

## 6.3 设计示例: 嵌入式数控系统

如图 6-4 所示,传统的嵌入式系统的设计将硬件和软件分为两个独立的部分分别进行设计,这种设计方法只能改善硬件/软件各自的性能,而不可能对整个系统做出很好的性能优化。因为,从理论上来说,每一个应用系统都存在一个适合于该系统的硬件和软件的最佳组合,依据一定的指导原则和分配算法对硬件/软件进行分析和合理的划分,从而使整个系统的性能达到最佳状态。因此嵌入式系统设计应是一个软、硬件结合的协同设计(Software/Hardware Co-design),如图 6-5 所示,这种协同设计需要硬件设计师和软件设计

师等有不同技术背景的人共同设计开发。

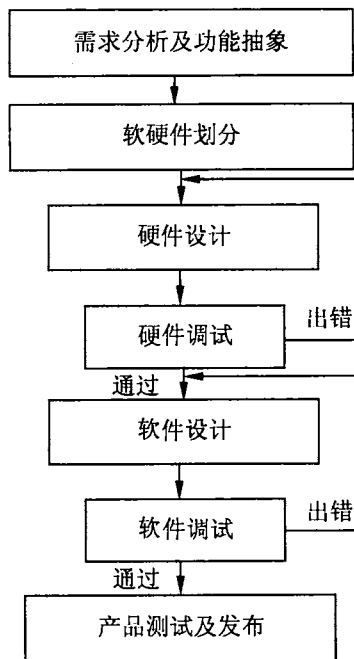


图 6-4 传统嵌入式系统设计采用的软件和硬件独立的设计方法

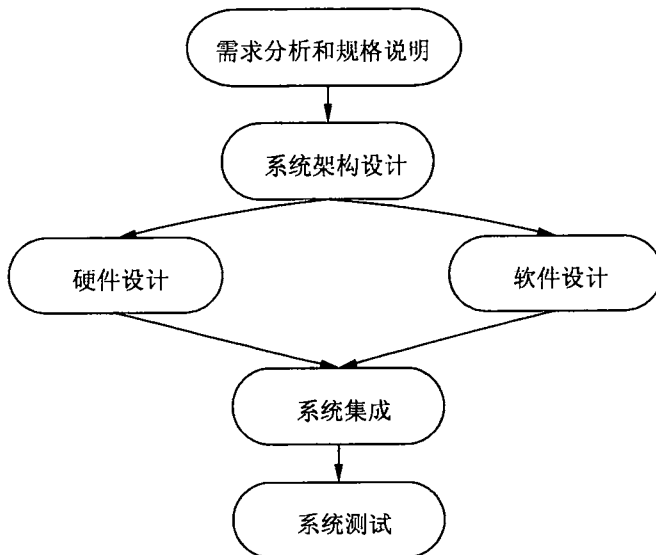


图 6-5 目前嵌入式系统所采用的软/硬件协同设计的方法

### 6.3.1 数控系统简介

数控系统是一种自动阅读输入载体上事先给定的数据，并将其译码，从而使机床移动和加工零件的控制系统。数控机床的整套控制装置由程序、输入输出设备、计算机数字控制装置、可编程逻辑控制器（Programmable Logic Controller, PLC）、主轴驱动装置和进给驱动装置等组成，习惯上称为 CNC 系统。图 6-6 所示是 CNC 系统构成框图，描述了系统内部的数据交换，其核心是计算机数字控制装置。从 20 世纪 50 年代数控机床问世至今，由于计算机技术和半导体技术的飞速发展，数字控制装置已成为计算机控制装置。数字控制装置采用微处理器和微型计算机的新技术后，其性能和可靠性大幅度提高，成本不断下降，优越的性能价格比推动了本行业快速的发展。

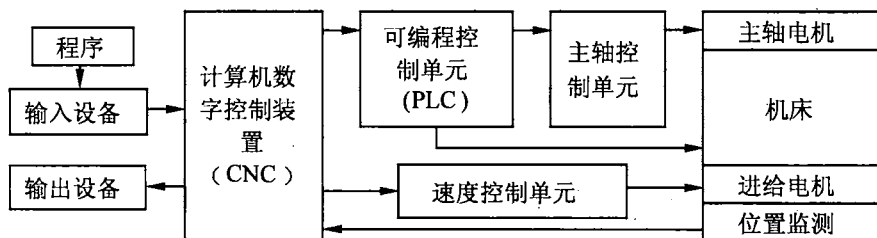


图 6-6 数控机床控制框图

由图 6-6 可知，系统内的控制由 CNC、PLC 完成。CNC 控制坐标的计算、插补和显示。程序的编辑、系统数据的输入和输出由人机交互界面完成，人机界面把用户输入的位置和速度信息输入 CNC 内部，控制装置接收指令控制机床移动，同时人机界面也负责反馈系统运行状态，使系统处于监控之中。

### 6.3.2 需求分析

#### 1. 功能需求

##### (1) 输入输出

- 键盘
- 机床 I/O
- 伺服驱动的输出
- 串行接口通信 RS232
- USB
- 字符图形显示
- 程序编辑

##### (2) 数据处理

- 译码功能
- 刀具补偿功能
- 速度处理功能
- 插补功能
- 主轴速度功能
- 刀具功能
- 辅助功能

#### (3) 报警

#### (4) 自诊断功能

### 2. 非功能需求

#### (1) 物理环境

车间，工作环境恶劣。

#### (2) 用户

一般是车间操作员，操作水平有高有低。

#### (3) 质量保证

用在工业控制环境质量要求高，同时现在市场竞争激烈，如果质量不好很难在市场上站住脚。

#### (4) QOS

- 数控系统响应性高，主要是对突发事件的反应（如撞刀，急停）。
- 数控系统具有可确定性。因为可确定性主要是确保条件/事件出现和由此引起的动作开始/结束的时间在一个准确的时间间隔内。在 CNC 系统中，条件/事件是由操作员的指令（紧急停止，移动 x 轴等）或是机床的状态（如刀具破损等）引起的。实际上，需要满足时间约束的情况主要是和系统安全（如对突发事件的反应等）以及切削精度（更高的精度影响插补周期）有关，因此数控系统具有硬实时任务。硬实时任务指必须满足最后期限的限制，否则会给系统带来不希望的破坏或者致命的错误。
- 性能高。需要进行许多复杂的运算
- 可靠性。可靠性要高，在加工过程中不出现问题，至少一个月之内不能死机，出现故障。
- 安全程度高。

### 6.3.3 系统体系结构设计

#### 1. 系统软硬件划分

根据数控系统所需完成的功能和需求，把数控系统分为 5 个任务，即人机界面管理任

务、数据处理任务、运动控制任务、逻辑控制任务，伺服控制，每个任务又可以划分为更小的子模块。人机交互为机床的准备工作提供数据和信息，反馈机床的运行状态，监控整个加工过程。数据处理主要包括数据指令的译码，刀具的长度补偿、半径补偿、螺距补偿、间隙补偿等插补前的预处理工作；运动控制主要控制位移、速度、加速度或三者的组合，主要是机床各运动轴的插补运动控制和主轴速度、主轴定位的控制等；逻辑控制分为简单的逻辑输入、逻辑输出及组合逻辑控制，主要是主轴电机的正反转、电机停止、冷却泵电机的启动、停止控制等；伺服控制是在给定的约束范围内各个轴执行运动指令所有必需的方法，这些指令通常是周期性的。

各个功能的软硬件分工如下：

#### (1) 人机界面

这些功能与运动控制没有直接关系，绝大多数使用软件来实现。除了键盘扫描功能以外。我们使用的 10×10 的扫描键盘，软件扫描程序的理论分析和实际测试的结果都表明，使用软件进行扫描是一项比较浪费微处理器运算资源的做法。在 66MHz 的 ARM7TDMI 微处理器加  $\mu$ C/OS-II 操作系统的平台上测试中发现，当使用软件进行 4×4 键盘的扫描时，一旦扫描任务的优先级高一些，会明显感觉到其他任务受到影响。因此，键盘扫描的任务将由 FPGA 来实现，当 FPGA 检测到有按钮按下时，向 CPU 发送一个中断。

#### (2) 逻辑处理

逻辑处理也就是 PLC 功能。PLC 是一项复杂的功能，涉及复杂的串行数学运算，因此决定使用 CPU 来实现。本系统定位在实现简单的 PLC 功能。辅助控制这部分功能较为繁琐，而且几乎与硬件没有任何关系，所以用微处理器来实现。

#### (3) 运动控制

运动控制是数控系统的核心，其中插补又是运动控制部分的核心。插补大体可以分为两级：粗插补和细插补。相对来说，粗插补负责将 G 代码转变为较详细的轨迹点信息，而细插补则将这些轨迹点细化为针对电机驱动器的脉冲信号输出。从数学运算上来看，粗插补的运算量很大，而细插补的运算量则很小。但是细插补对时间的准确性要求非常高，如果要用微处理器实现，则需要一个周期非常小的定时器，而且周期也会不断变化，这样会消耗大量的微处理器计算时间，甚至微处理器没有时间运行其他的任务。对于 FPGA 来说，实现这样的细插补功能则非常简单，只需要很少的逻辑资源，而且脉冲的最大频率也可以很高，至少可以远远超过外部接口元件的极限。因此，粗插补将由微处理器负责，而细插补理所当然由 FPGA 来实现。

#### (4) 数据处理

这部分功能也几乎与硬件没有任何关系，所以用微处理器来实现。

#### (5) 伺服处理

与运动控制的情况类似, 伺服算法的实现需要大量的串行的数学运算, 而信号检测部分如果用软件实现的话则需要微处理器不断去检测信号的变化。因此, 伺服算法部分由微处理器实现, 信号检测部分由 FPGA 实现。

## 2. 硬件系统划分

虽然如上一小节所说, 很多功能是软件实现的, 但是软件也是运行在硬件上的, 所以说, 在进行硬件系统的划分时也必须把软件的运行基础考虑在内。这样, 从硬件设计者的角度上去分析, 整个电路板系统可以分为板级系统和芯片级系统。

板级系统的设计指印制电路板 (PCB) 的设计, 芯片级系统指 FPGA 内部逻辑的设计。两者虽然都是属于硬件的范畴, 但是两者的设计对象、设计方法、开发流程和所需要使用的 EDA 软件都完全不同。

板级系统由微处理器子系统、FPGA 子系统、D/A 转换子系统、信号隔离与转换子系统、电源子系统构成。微处理器子系统负责运行数控的控制软件, FPGA 子系统负责脉冲信号的产生和计数、键盘的扫描和 I/O 的控制, D/A 转换子系统负责产生主轴变频器所需要的模拟信号, 信号隔离与转换子系统负责各类机床信号的接口处理, 电源子系统则为其其他系统和继电器提供电源。结构框图如图 6-7 所示。

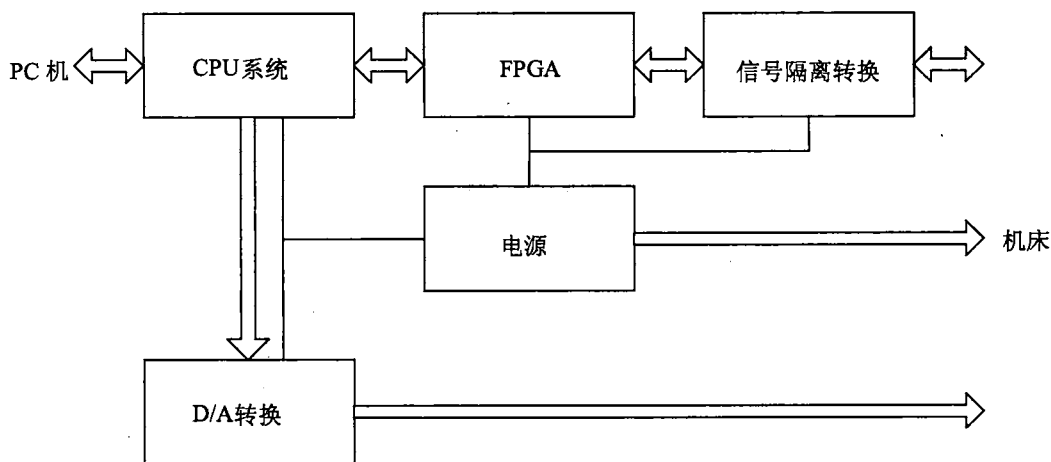


图 6-7 板级系统框图

芯片级系统由总线接口模块、复位控制模块、中断控制模块、定时器模块、I/O 控制模块、键盘扫描编码器计数器模块和驱动器控制器模块构成。其中总线接口模块负责提供 FPGA 内部功能模块与 ARM 外部总线的接口, 复位控制模块为 FPGA 内部功能模块提供复位信号, 中断控制模块用于处理 FPGA 内部功能模块的中断信号, 定时器模块为脉冲发生器提供定时信号, I/O 控制模块用于控制顺序逻辑 I/O, 键盘扫描模块负责控制 10×10 键

盘的扫描,编码器计数器模块用于检测主轴和手轮的码盘信号,驱动器控制器模块用于产生、检测电机驱动器的信号。整个结构框图如图6-8所示。

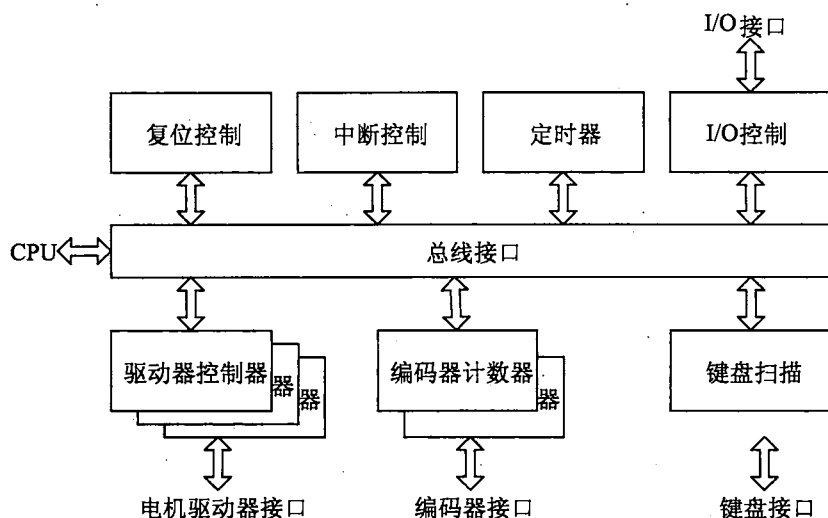


图 6-8 芯片级系统框图

### 3. 系统软件功能划分

由于数控系统较为复杂,编码和测试工作量都很大,开发过程中进行了详细的任务划分,软件的开发采用模块化开发,多人分工同时进行。由操作系统管理的每一个任务并不能简单地划分为开发中的一个模块,因为每个任务之间并不能保证相对的结构独立,更不能保证每个任务的子函数无交叉调用,因此,必须重新划分开发中相对独立的模块。根据数控系统每部分的物理意义和编码过程中的相对独立性,以及测试和仿真的方便性,本系统开发中划分为以下几个模块:人机界面管理任务、数据处理任务、运动控制任务、逻辑控制任务、伺服控制任务。

## 6.3.4 硬件设计

### 1. 板级设计

#### 1) 板级设计的原则

##### (1) 模块化设计

无论是原理图设计,还是PCB设计,都要遵守模块化设计的思想。原理图模块化可以使设计思路清晰,原理易于掌握,差错容易。PCB模块化则对于调试和可靠性的提高都有很大好处。而且模块化设计对于系统以后的维护、升级都有很多方便。



### (2) 可重构原则

本系统采用的 FPGA 要具有在系统可编程 (In System Programmable,ISP) 的能力,因此当一套产品生产出来以后,还可以改变 FPGA 内部的逻辑从而实现 FPGA 功能的改变。CPU 系统也要具有在系统烧写 Flash Memory 的能力,这样可以在不动元件的情况下改变存储的程序,从而改变软件系统的功能。

此外,为了将来可能出现的较大规模的升级,例如添加电机控制闭环功能,或者要求能够多控制一台驱动器,还需要将 FPGA 的一部分备用 I/O 功能的接口电路用背板的形式处理,这样将来需要使用这些 I/O 接口的时候,可以只添加一块新的接口电路背板,而不需要重新设计整个主板。

### (3) 兼容性原则

由于本项目的特点,为了加快开发速度和便于以后生产,要尽量按照 980T 的现有部件,如机箱、开关电源、LCD、接口电路的元件、接口插件等的标准去设计,也就是除了控制核心以外,其他的部分都要尽量与 980T 系统兼容。这样设计出来的系统可以直接使用现有的部件和硬件测试系统,能够大大的方便后期的调试和产品化工作。

## 2) 性能分析与初步设计

### (1) CPU 计算能力

虽然 CPU 的计算能力可以比较准确的估计,但是由于实际使用时的算法不同,所以估计结果与最终体现出来的结果会有很大差距。由于以前没有过具体的测试,所以只能根据现有系统的计算能力与实际实现结果的资料推测。

目前的 980T 系统采用的是 16 位的 CPU—Z180, CISC 结构,主频为 20MHz,可以控制两个轴的运动和大约 40 个 I/O 的顺序动作。同时完成对 8×8 左右规模的键盘的扫描和 LCD 的控制。在这些工作中,首先 I/O 顺序控制几乎不需要什么运算量,然后 LCD 采用外部的扫描控制芯片,所需 CPU 的运算极少,最后是键盘的扫描。在我们所做的  $\mu$ C/OS2 系统中,键盘扫描是作为一个任务实现的,实际使用的结果发现,还是需要耗费不少 CPU 的运算量的。但是在原先的系统没有使用操作系统,节省了很多任务切换和内部信号量处理时间,所以应该不会耗费多少 CPU 的运算资源。这样,绝大多数运算资源是用来实现两个轴的运动控制的,而且全部都是 4 字节或者 8 字节的运算。

现在使用 32 位的 CPU 替代原有的 CPU,具体型号为 S3C44B0X。该 CPU 属于 ARM7TDMI 结构, RISC, 最高速度 66MHz。由于数控系统的工作条件比较恶劣,所以假设降频至 20 kHz 使用。首先,由于 S3C44B0X 是 32 位 CPU,内部有 8KB 缓存,所以使用缓存时,仅从位数上来说,运算能力差不多是原先 CPU 的两倍。但是要考虑如下几个不利因素:

- ARM7TDMI 是 RISC 结构,代码效率不如 CISC,也就是相同频率、相同位数的条

件下, 在运行一般程序的时候, CISC 系统可以完成更多的运算。

- S3C44B0X 内部带 LCD 控制器, 当 LCD 刷新时需要 DMA 操作, 这是就会占用数据总线, 从而影响 CPU 的运算。
- 设想中使用操作系统, 操作系统通常占 CPU 时间资源的 3%~5%。

以及如下几个有利因素:

- ARM7TDMI 带有部分 DSP 指令, 这几条指令对于数控这样需要大量数学运算的应用十分有用。
- 由于可寻址的存储器大大增加, 编程时可以根据速度要求优化。
- ARM7TDMI 可以达到 0.9 MIPS/MHz, 几乎都是单周期指令。

综合以上几个因素, 估计数学运算能力, 尤其是需要 DSP 运算的数学运算, ARM7TDMI 至少可以达到 2 倍于 16 位 CPU 的能力。而对于布尔运算和位运算, 由于 ARM7TDMI 的 0.9 MIPS/MHz, 优势会更明显。因此, 单从数学运算角度上分析, 可以控制 4 个轴的运动。

这些都是在 20MHz 的条件下估计的, 如果将来发现 CPU 还是不够用, 可以提高主频。

## (2) 实时性

实时性主要包括响应时间和响应时间是否固定两个方面。首先, 对于 LCD 的刷新问题, 当时用 DMA 方式时, 在 DMA 传送过程中不能响应中断, 所以将不采用 DMA 方式。又由于 ARM7TDMI 的中断系统复杂, 所以中断响应会比 Z180 慢, 但是肯定可以满足实时性的要求。但是, 实际使用时是带操作系统的, 这样就决定了是否实时很大程度上依赖于操作系统的实时性。 $\mu\text{C}/\text{OS}2$  是专门用于控制领域的实时操作系统, 它的中断响应时间是固定的。因此, 问题集中在响应时间上。当采用不同的软件结构的时候, 响应时间是不一样的, 这里不想过多讨论软件问题, 仅从硬件上讨论如何让软件有一个良好的运行基础。

本系统的实时性最强的任务就是电机的运动控制。在本系统中, 使用 FPGA 产生脉冲信号。这个方法与 980T 使用 CPLD—MAX7128 产生信号的方法是一样的, 但是现在是使用 Acex 系列 FPGA, 其逻辑容量比 MAX7128 有很大的提高。利用大量的逻辑资源可以实现很多复杂的逻辑。首先就是实现脉冲数据的缓冲。由于电机控制的实时性要求, 当所需要发的脉冲发完以后再向 CPU 中断, 然后 CPU 再发送新一组数据, 显然不合理, 而且如果 CPU 正在中断处理之中, 无法及时响应就更麻烦了。所以在 FPGA 内部添加一级缓冲, 也就是当 FPGA 在按照 CPU 给定的数据发送脉冲的时候, CPU 可以向 FPGA 写入下一次要发送的脉冲的数据, 这样, 当 FPGA 发送完本次的数据后可以立即开始发送下一个数据, 可以做到绝对的实时。

由于有 FPGA 的缓冲, CPU 的实时性要求大大降低, 对于软件的规划有很大方便之处。

## (3) 存储能力

32 位的 S3C44B0X 与原先的 16 位 CPU Z180（其数据、程序存储器各 64KB）相比，程序和数据存储能力的扩充非常大。S3C44B0X 的程序、数据存储范围是 256MB，但是由于某些地址被内部占用，还有一些受地址总线的限制，实际所能扩充的可寻址范围要小一些。而且 S3C44B0X 可以使用 SDRAM，大大提高了存储密度，降低了成本。SDRAM 最多可以扩充到 64MB。这对于高级一些的应用足够了。同时，在系统中还有 NAND Flash，这是类似电子盘的元件，容量为 16MB，在 PCB 不变的情况下可以有 128MB 容量。

#### （4）FPGA 的选择和 I/O 扩展能力

ACEX1K 是 Altera 公司的基于 SRAM 技术的、用于低端应用的高性价比 FPGA，分为 1K10、1K30、1K50 和 1K100，容量分别为 1 万、3 万、5 万、10 万门，虽然容量不同，但是在相同封装的情况下引脚兼容（个别情况除外）。

根据需求，顺序控制 I/O 需要 72 个，键盘需要 20 个，电机驱动器接口需要 18 个，主轴、手轮需要 5 个，共计 115 个。显然 S3C44B0X 自身的 I/O 接口线显然不能满足要求。本系统采用 FPGA 来扩展 I/O。采用 Acex 的 1K50 或者 1K100，208 引脚 PGFP 封装的芯片，共有 147 个用户可用 I/O 接口。其中，部分用于实现与 S3C44B0X 的通信，需要大约 30 个。这样，剩余的 I/O 接口可以满足需要。

为了调试需要，样机会采用 1K100，一旦电路成熟以后可以根据实际需要采用较小容量的芯片。

#### 3) 板级功能划分

##### （1）微处理器子系统

微处理器子系统包括 ARM 子系统、存储器子系统和 LCD 接口、通信接口和串行接口，组成框图如图 6-9 所示。

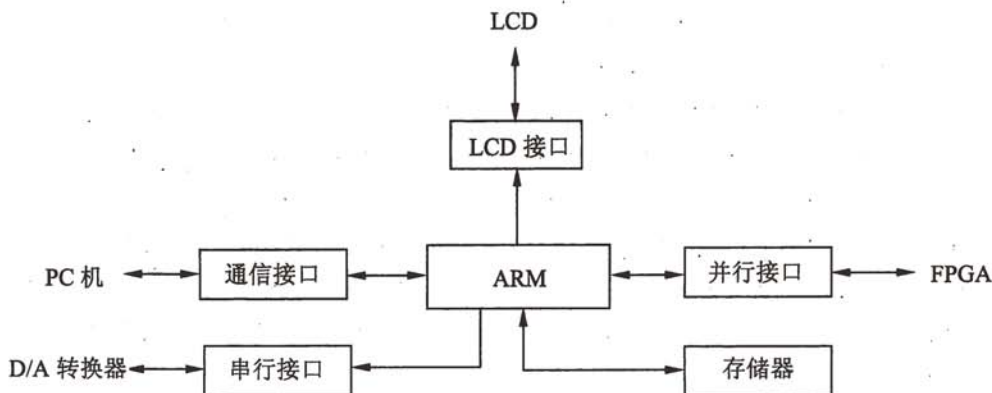


图 6-9 微处理器子系统框图

其中, ARM 是一种嵌入式处理器, 主要负责运算; 存储器负责程序和数据的存储以及文件系统; 通信接口负责加工程序的下载和上传; 并行接口实现与 FPGA 的通信; 串行接口实现对 D/A 转换器的控制。

存储器子系统包括 Flash、SDRAM 和 NVRAM。Flash 存储程序 and 文件，SDRAM 存储系统运行时的程序和数据，NVRAM 存储实时的系统状态。

通信接口实现与 PC 机的基于 RS232 标准的数据通信, 并行接口实现与 FPGA 的数据通信, 串行接口实现对 D/A 转换器控制。

## (2) FPGA 子系统

FPGA 子系统包括 FPGA、配置电路、下载接口、并行接口，组成框图如图 6-10 所示。

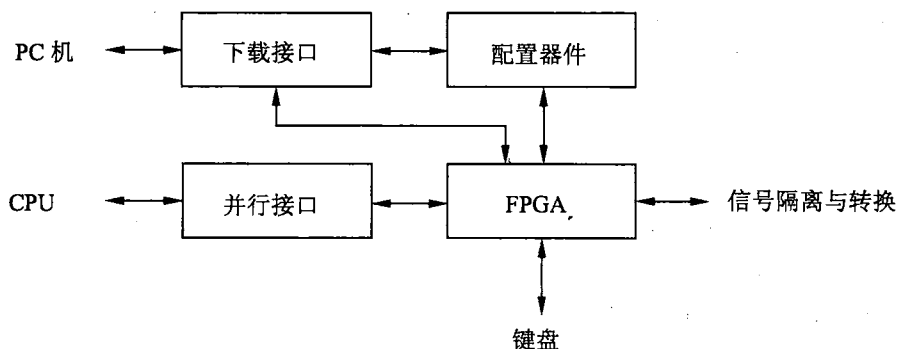


图 6-10 FPGA 子系统框图

配置元件用于在上电时配置 FPGA，下载接口用于烧写配制元件和直接配置 FPGA，并行接口实现与 CPU 子系统的通信。

### (3) D/A 转换子系统

D/A 转换子系统由隔离元件、D/A 转换器和运放组成，框图如图 6-11 所示。

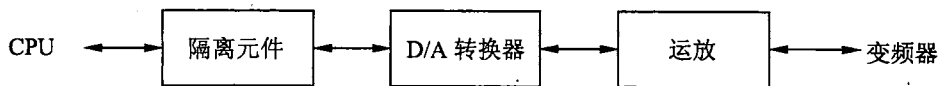


图 6-11 D/A 转换子系统框图

隔离元件在 CPU 和 D/A 转换器之间,属于数字隔离方式,与使用模拟光耦或者隔离运放相比较,可以降低成本、提高精度,虽然实际系统模拟输出的刷新速度会有一些影响,但是对于本系统还是可以满足要求的。

#### (4) 电源子系统

新型的 CPU 和 FPGA 的内核电压一般都使是 2.5V 或以下的, I/O 电压一般都使是 3.3V, 高速光耦一般也需要 5V 电压。根据要求, 要使用现有的 980T 系统的开关电源, 该电源可以提供 5V,  $\pm 12V$  和 24V 电源。因此, 需要电源模块使用 5V 产生 3.3V 和 2.5V, 为了隔离的需要, 还要使用 24V 产生另外的一个 5V 电压。

#### 4) 实现方案

根据需求, 主要元件的选择方案如下: CPU 采用 S3C44B0X, 存储系统包括 2MB 的 Flash—AM29LV160, 用于存储程序; 32MB 的 SDRAM—HY57V561620, 用于存储运行时的程序和数据; 32KB 的 NVRAM—DS1230, 两片, 用于实时保存数据; 16MB 的 NAND Flash—K9F2808U, 用于实现文件系统; FPGA 采用 208 引脚的 EP1K50, 配置存储器使用 EPC2; D/A 转换器采用 AD7243, 0~10V 输出; 电源模块采用两片 LM1085 和一片 LM2575, 分别用于产生各种电压。

#### 5) 具体实现

##### (1) 微处理器子系统

##### ① ARM 子系统

ARM 子系统包括 S3C44B0X 型嵌入式 CPU、时钟、复位电路、JTAG 接口和实时时钟 (Real Time Counter, RTC) 电路。

S3C44B0X 除了具有 ARM7TDMI 内核以外, 还具有多种片内外围设备, 主要包括存储器接口、LCD 控制器、异步串行接口、同步串行接口、通用 I/O 接口等。

时钟采用的是 6MHz 外部晶体, 利用 ARM 片内的 PLL 模块, 可以将 CPU 的运行速度提高到 66MHz。

复位电路没有使用普通的阻容复位, 而是采用的复位专用芯片 IMP811T, 该芯片具有电压监视和手动复位输入功能。当供电电压小于 3.08V 时输出复位信号。

JTAG 接口按照 ARM 公司的建议电路设计, 如图 6-12 所示。要特别说明的是由于 ARM7TDMI 内部 JTAG 的设计不是实现通过 JTAG 接口控制整个芯片的复位, 所以这里把系统的复位信号通过一个跳线与 nTRST 相连, 这样可以让 S3C44B0X 里的 JTAG 状态机与其他部分同时复位。

RTC 电路用于在掉电时维持 S3C44B0X 内的实时时钟电路的运行, 经过软件的初始化以后, 该电路可以为系统提供准确的绝对时间。RTC 电路的电源部分比较重要, 为了长久地使用 RTC 功能, 必须要给在掉电的时候为 RTC 电路供电的电池充电。RTC 电路的电源部分的电路如图 6-13 所示,  $V_{DD33}$  为 3.3 V,  $V_{DDRTC}$  为 2.5 V, 是 RTC 的电源,  $V_{DD5}$  为 5 V。

由于二极管的正向压降与所通过的电流有关, 在实际测试中发现, 当电池处于电量满的时候, 上电前 DD2005 和 DD2003 总共的压降为 0.8V 左右, 这样  $V_{DDRTC}$  可以稳定在

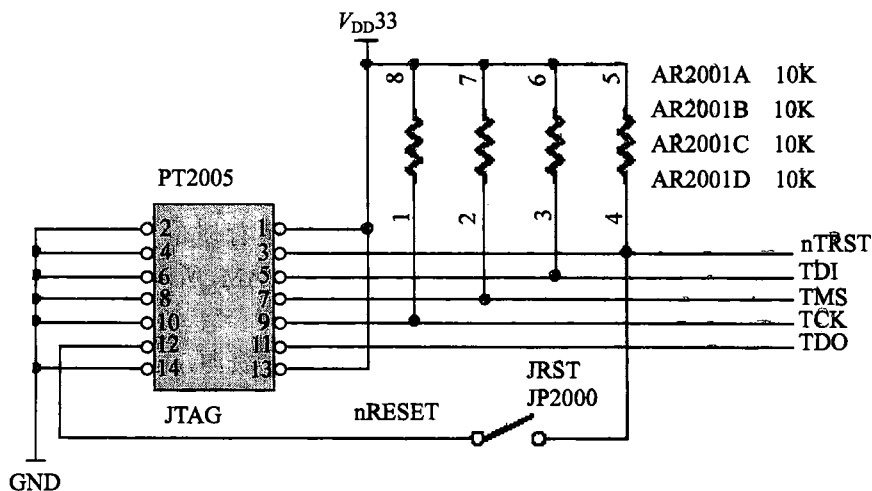


图 6-12 JTAG 接口

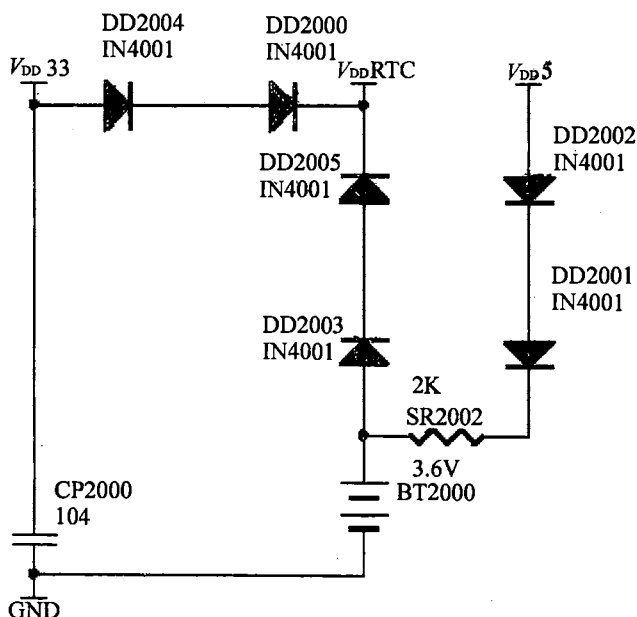


图 6-13 RTC 供电电路

2.8V。上电后每个 DD2004 和 DD2000 总共的压降为 0.2V 左右，这样  $V_{DDRTC}$  可以稳定在 2.9V。仍然处于 S3C44B0X 数据手册上的说明的工作电压范围之内。

## ② 存储器子系统

NOR Flash 采用 Am29LV160B, 2MB, 用于存储启动代码, 该代码完成对 S3C44B0x 的初始化, 然后将主程序从自身复制到 SDRAM 里。实际存储容量需要根据主程序存放的位置不同, 相差较大, 但是为了调试采用了较大的。由于 S3C44B0x 复位后自动从 0x00000000 开始执行, 所以该 Flash 使用了 BANK0。

SDRAM 采用 HY57V561620, 32MB, 用于存储运行时的主程序和常量。由于 S3C44B0X 的限制, 只有 BANK6 和 BANK7 可以使用 SDRAM, 所以, 该芯片使用 BANK6, 起始地址是 0xC0000000。

NAND Flash 采用 K9F2808U, 16MB, 用于存储文件和资源, 使用 BANK1。由于非线性 Flash 的特点, 不是直接的总线访问, 而是类似硬盘的命令访问, 所以还用 4 个 I/O 接口用于控制。

NVRAM 采用 DS1230W, 16KB, 用于存储机床的状态, 防止突然掉电时损失数据。

## ③ 通信接口

RS232 电平转换使用的是 MAX3232, 兼容 3 V~5.5 V 的电平, 可以直接与 3.3 V 的 S3C44B0X 连接。

## ④ LCD 接口

LCD 接口由控制信号接口电路和 24V LCD-驱动电压产生电路组成。LCD 采用的 LCD 是 Lm32019T, 其控制信号包括开关信号、数据信号和同步信号。开关信号使用一个普通的 I/O 信号, 同步信号有 VFRAME-帧同步信号, VLINE-行同步和 VCLK-列同步信号, 这 3 个是必需的控制信号, 数据信号使用 VD0~VD3。由于 Lm32019T 的数字部分都是 5V 逻辑, 因此, 需要一片 74LS245 进行电平转化。

24 V LCD 驱动电压产生电路主要包括升压芯片和可编程电位器。使用 CPU 的 3 个 I/O 输出信号可以实现对可编程电位器的控制。

## (2) FPGA 子系统

### ① 配置电路和下载接口

配置芯片和下载配置电路用于使用专用的下载线——ByteBlasterMV 对配置芯片进行烧写, 对 FPGA 进行在线配置, 配置芯片对 FPGA 进行配置。其中 ByteBlasterMV 对配置芯片进行烧写和对 FPGA 进行在线配置使用的是串行 JTAG 链, 而配置芯片对 FPGA 进行配置使用的是专用的配置电路。这样, 可以在不改变硬件的情况下同时对配置芯片和 FPGA 进行控制, 也可以在烧写配置芯片以后在加电时由配置芯片配置 FPGA, 大大方便了调试和使用。

配置芯片使用的是 EPC2, 该芯片是实际上是串行 Flash, 通过 JTAG 接口进行烧写, 通过专用信号配置 FPGA。配置电路和下载接口的电路如图 6-14 所示。

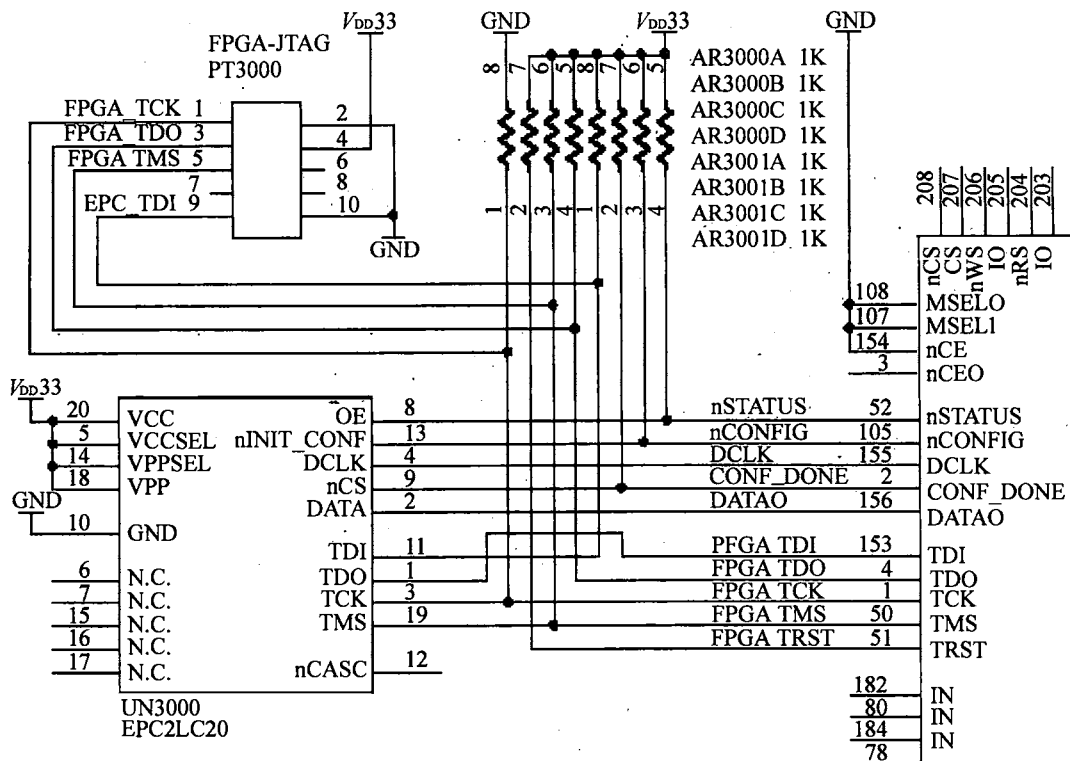


图 6-14 配置电路和下载接口的电路。

## ② 并行接口

FPGA 使用 S3C44B0X 的外部总线与 S3C44B0X 进行通信, 需要用到的 S3C44B0X 的信号有: 16 根数据线、8 根地址线、读、写、片选、中断。这样总共有 512B 的访问空间。

## (3) D/A 转换子系统

### ① 隔离

使用高速光耦 6N137 实现隔离, 不需要复杂的设计, 6N137 的速度就可以达到 1 MB, 传递 16 位数据只需要 16  $\mu$ s。需要隔离的信号有 3 个: 同步、时钟、数据, 而且都是单向的, 所以只用 3 只 6N137 就够了。由于光耦的输入端电流较大, 直接使用 S3C44B0X 的引脚会对 S3C44B0X 有较大的电流冲击, 所以先用 74HC245 进行驱动, 然后控制 6N137。隔离电路如图 6-15 所示

### ② 转换

D/A 转换器采用 AD7243, 这是一款 12 位串行 D/A 转换器, 具有多种电压输出范围和多种工作模式, 如图 6-16 所示。



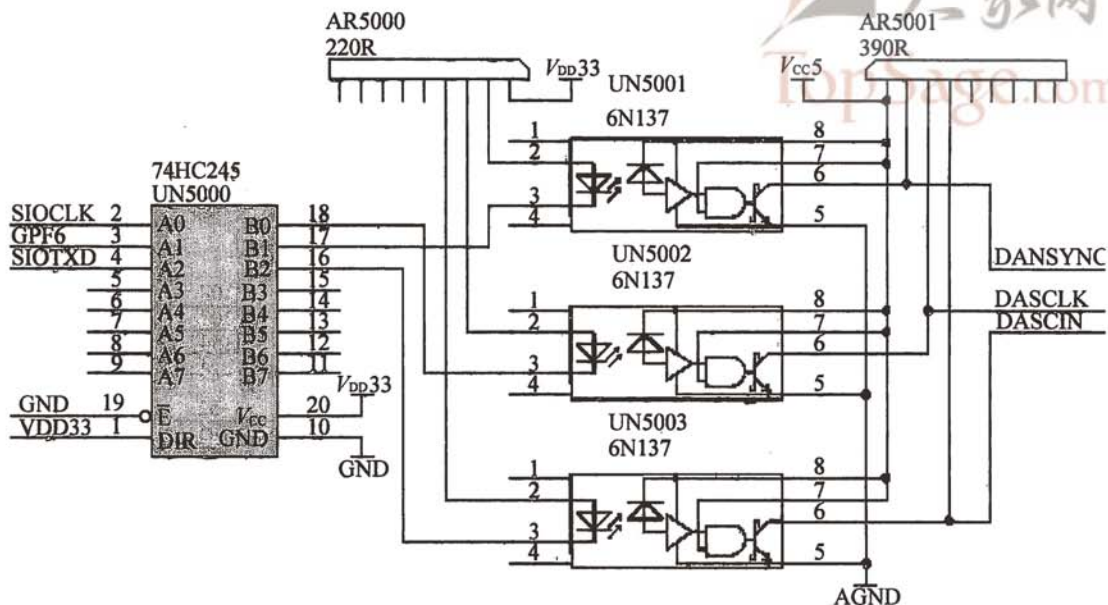


图 6-15 D/A 转换隔离电路

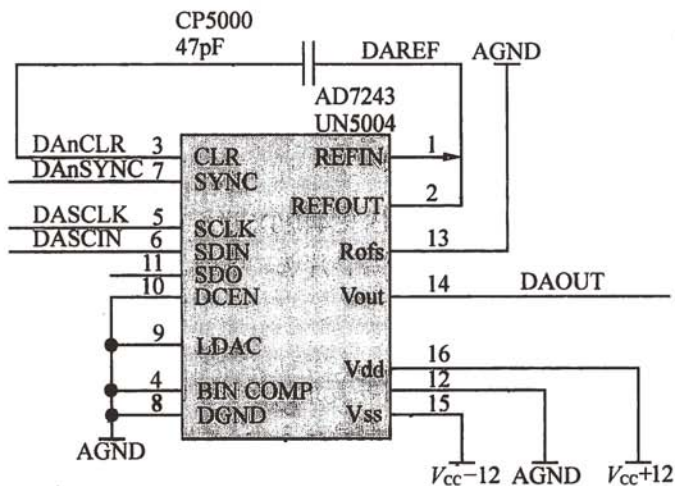


图 6-16 D/A 转换隔离电路

### ③ 放大

为了提高模拟输出的阻抗性能,需要使用运算放大器进行处理。由于 D/A 转换器输出是 0~10 V 电压,所以采用射随器电路。运放使用 MC4558,是因为在 980T 系统上就是采

用的 MC4558。模拟输出运放电路如图 6-17 所示。

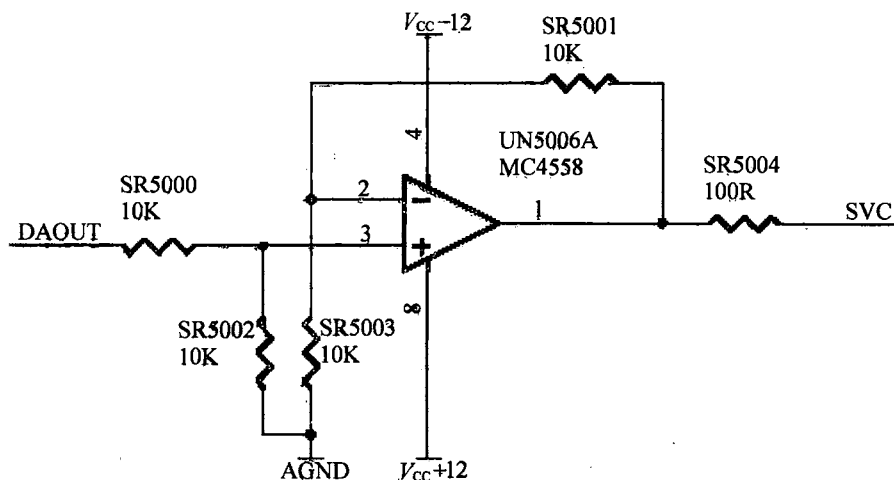


图 6-17 模拟输出运放电路

#### (4) 电源子系统

可以选择元件的电源模块有线性稳压器和开关电源模块两种。线性稳压器的特点是输出品质好，外电路简单，但是当输入输出电压差较大时，输出功率会受到很大影响。而开关电源模块输出功率受输入输出电压差影响较小，但是品质差一些，外部电路也比较复杂。

因此，使用线性稳压器实现 5V 和 3.3V 以及 5V 和 2.5V 之间的转换，使用开关电源模块实现 24V 和 5V 之间的转换。线性稳压其使用 LM1085，可以提供 1A 的输出电流。开关电源模块使用的是 LM2575-5，效率可以达到 80%，电路如图 6-18 所示。

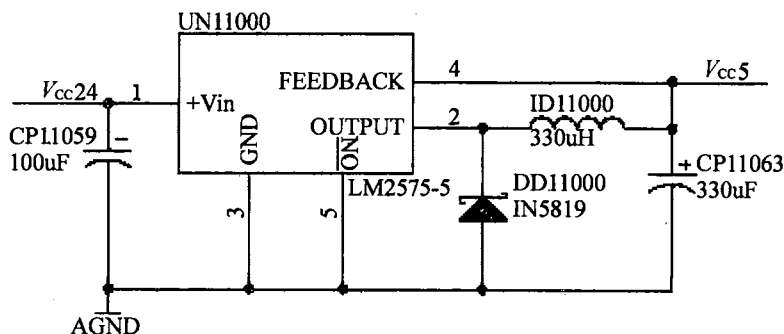


图 6-18 24 V 和 5 V 之间的转换电路

## 2. 芯片级硬件设计

### 1) FPGA 介绍

FPGA（现场可编程门阵列）是一种可编程逻辑元件，它是在 PAL 和 GAL 等逻辑元件的基础之上发展起来的。同以往的 PAL 和 GAL 等相比较，FPGA 的规模大得多，而单位逻辑门的成本却低得多。大容量、低成本为 FPGA 在数控系统的应用创造了条件。利用 FPGA 可以实现 I/O 处理，脉冲发生、计数，数学运算等功能，可以大大简化数控系统的设计。

### 2) FPGA 的开发

目前 IC 设计主要使用的硬件描述语言（Hardware Description Language, HDL）和原理图进行逻辑描述。随着 EDA 技术的发展，使用硬件语言进行设计正成为一种趋势。HDL 有很多种，目前最流行的是 VHDL（Very-high-speed integrated circuit Hardware Description Language）和 Verilog HDL。相对来说，VHDL 语法严格、书写规则繁琐，而 Verilog HDL 是在 C 语言基础上发展起来的，语法较自由，但是也容易让少数初学者出错。

HDL 和传统的原理图输入方法的关系就好比是高级语言和汇编语言的关系。HDL 的可移植性好，使用方便，但效率不如原理图；原理图输入的可控性好，效率高，比较直观，但设计大规模电路时显得很烦琐，移植性差。本系统的 FPGA 逻辑全部是用 VHDL 编写的。

### 3) 开发流程与 EDA 软件

用 HDL 开发 FPGA 的完整流程为：

（1）文本编辑。也就是编写代码，可以使用一般的文本编辑器。

（2）功能仿真。使用 HDL 仿真软件对代码文件进行功能仿真，检查逻辑功能是否正确，也叫做前仿真。

（3）综合映射。使用逻辑综合软件对代码文件进行综合，即先把语言描述综合成最简单的布尔表达式和信号的连接关系，然后根据 FPGA 的结构生成 FPGA 内部标准单元之间的连接关系。逻辑综合软件会生成网络表文件——EDIF（Electronic Design Interchange Format）文件。

（4）布局布线。使用 FPGA 厂家提供的布局布线软件对逻辑综合软件生成网络表文件进行布局布线，生成下载文件和实际的延时信息文件，即把设计好的逻辑安放到 FPGA 内。

（5）时序仿真。用 HDL 仿真软件结合代码和延时信息文件验证电路的时序，也叫做后仿真。时序仿真可以真实地反应实际的工作情况。

（6）下载验证。使用 FPGA 厂家提供的下载工具将下载文件下载到芯片中，观察运行结果，看看是否满足需要。

目前主要的 FPGA 厂商提供完整的开发环境，可以完成上述所有的工作。但是在仿真、综合两个步骤里与专门的软件开发厂商提供的工具相比性能往往比较差，例如对 HDL 支

持不够好, 仿真速度慢, 综合效果差等。因此, 建议采用专业的软件。

目前最好的 PC 机仿真工具是 ModelSim、VCS、NC-SIM 和 Active-HDL, 综合工具是 Synplify/Amplify、FPGA Compiler 和 LeonardoSpectrum。开发本系统使用的是 Active-HDL 和 Amplify。

### 6.3.5 软件设计

#### 1. 软件接口设计

##### 1) $\mu\text{C}/\text{OS-II}$ 实时操作系统

$\mu\text{C}/\text{OS-II}$  是开源代码的面向控制的操作系统, 具有可固化 (ROMable)、可裁剪 (Scalable)、占先式 (Preemptive)、多任务、可确定性、稳定可靠的特点。

数控系统作为一个结构复杂、功能繁多的系统, 使用  $\mu\text{C}/\text{OS-II}$  最大的好处就是利用它的多任务特性, 可以将界面、文件系统、译码、插补、伺服等功能使用单独的任务完成, 这样可以大大方便软件开发过程中的项目管理工作。同时, 各个任务之间相互独立, 当一个任务出现问题时不影响其他任务, 还可以用其他任务来关闭出错的任务, 并重新初始化该任务, 增强了系统的可靠性。

但是任何事务都是有利也有弊的。 $\mu\text{C}/\text{OS-II}$  是一个占先式的内核, 即已经准备就绪的高优先级任务可以剥夺正在运行的低优先级任务的 CPU 使用权。这个特点使它的实时性比非占先式的内核要好。通常都是在中断服务程序中使高优先级任务进入就绪态 (例如发信号), 这样退出中断服务程序后, 将进行任务切换, 执行高优先级任务。拿 8C51 单片机为例, 比较一下就可以发现这样做的好处。假如需要用中断方式采集一批数据并进行处理, 在传统的编程方法中不能在中断服务程序中进行复杂的数据处理, 因为这会使中断时间过长。所以经常采用的方法是置一标志位, 然后退出中断。由于主程序是循环执行的, 所以它总有机会检测到这一标志并转到数据处理程序中去。但是因为无法确定发生中断时程序到底执行到了什么地方, 也就无法判断要经过多长时间数据处理程序才会执行, 中断响应时间无法确定, 系统的实时性不强。如果使用  $\mu\text{C}/\text{OS-II}$ , 只要把数据处理程序的优先级设定得高一些, 并在中断服务程序中使它进入就绪态, 中断结束后数据处理程序就会被立即执行。这样可以把中断响应时间限制在一定的范围内。对于一些对中断响应时间有严格要求的系统而言, 这是必不可少的。但因为  $\mu\text{C}/\text{OS-II}$  要求在中断服务程序末尾使用 `Osintexit` 函数以判断是否进行任务切换, 这需要花费一定的时间。而在  $\mu\text{C}/\text{OS-II}$  的中断处理函数内, 很多系统函数是不能使用的, 因此对于某些必须在中断内直接进行处理的场合, 就必须使用额外的方法来解决某些系统函数不能使用的问题。

##### 2) 引导结构

虽然  $\mu\text{C}/\text{OS-II}$  具有可固化的特性, 但是为了便于调试, 在本系统中还是设计了一个两

级引导的结构。固化在 Flash 起始地点的是引导程序，它负责基本的硬件初始化，然后将硬件检测系统加载到 SDRAM 中运行。硬件检测系统运行以后，首先检测串行接口输入，如果在规定的时间内没有输入，那么将数控系统程序加载到 SDRAM 中运行。

由于硬件检测系统比较小，因此加载到 SDRAM 的高 16MB 中，而数控系统程序加载到 SDRAM 的低 16MB 中，因此只要数控系统程序的 ROM 代码不大于 16MB 就不会出问题。

虽然硬件检测系统和数控系统程序的功能不同，但是它们都是使用的同一个代码树，只是宏定义不同。也就是说都是使用  $\mu\text{C}/\text{OS-II}$ ，保证了任何对软/硬件接口的改变会同时作用在两套系统上，只要经过硬件检测系统的验证以后就可以直接用在数控系统程序。而且两套系统都具备硬件初始化的功能，这样在实际产品中可以只使用数控系统程序，不再需要引导程序和硬件检测系统。

### 3) 硬件检测系统

硬件检测系统是一个基于串行接口和命令行的对硬件进行测试的工具。使用 PC 机或者其他具备 RS232 接口的主机都可以与数控系统相连，通过运行在 PC 机或者其他主机上的终端程序，可以以命令行的方式运行各种命令，并查看运行的结果。

由于很多重要的硬件功能的实时性很强，因此不能通过仿真器观察运行的结果，只能通过串行接口显示或者存储在缓冲区里在通过串行接口显示的方法。很多功能复杂的软件模块的运行连续性要求也很强，它们的测试也必须使用这个方法，比如在一个工具的加工测试中，为了测试系统运行的状态，必须保持系统的运行，不能在加工的过程中停止运行然后观察结果。

除了可以进行硬件的检测以外，硬件检测系统还可以使用 XModem 协议传送文件，是数控系统和主机传递数据的唯一方法。

如前所述，硬件检测系统和数控系统程序使用相同的代码树，因此大多数在硬件检测系统里可以执行的代码在数控系统程序里也可以执行，只有涉及电机运动的指令是不能运行的。

还需要特别指出的是，硬件检测系统里的重要指令——加载数控系统程序，在加载数控系统程序本身里也是可以使用的。当系统的某些 ROM 代码被意外破坏之后，可以使用加载数控系统程序指令重新启动系统。如果在启动之前做好足够的准备，还可以最大限度地还原到问题发生以前的状态。

### 4) 数控系统程序接口

#### (1) FPGA 接口

除了 D/A 输出功能以外，其他所有的数控硬件功能都是由 FPGA 最终控制的，因此，数控系统程序必须能够对 FPGA 进行控制。如前所述，为了提高模块化、维护性、可读性，

所有对 FPGA 的控制都可以通过函数来实现。而这些函数依靠对 FPGA 的读写完成实际的操作。表 6-2 所示是 FPGA 寄存器地址分配表。

表 6-2 FPGA 寄存器地址分配表

地 址	名 称	功 能
0x08000000	IDL	芯片版本、序号描述寄存器低 16 位
0x08000002	IDH	芯片版本、序号描述寄存器高 16 位
0x08000004	RST	复位控制寄存器
0x08000006	SYS	系统控制寄存器
0x08000008	INT	中断系统控制寄存器
0x0800000A	TMRCON	定时器控制寄存器
0x0800000C	TMRDAT	定时器周期寄存器
0x0800000E	IOINL	I/O 输入状态寄存器, 对应 IOIN 的 0~15
0x08000010	IOINH	I/O 输入状态寄存器, 对应 IOIN 的 16~31
0x08000012	IOOUTL	I/O 输出控制寄存器, 对应 IOOUT 的 0~15
0x08000014	IOOUTM	I/O 输出控制寄存器, 对应 IOOUT 的 16~31
0x08000016	IOOUTH	I/O 输出状态寄存器, 对应 IOOUT1 的 32~39
0x08000018	KEYCON	键盘扫描控制寄存器
0x0800001A	KEYSCP	扫描周期寄存器
0x0800001C	KEYDNP	去掉延时寄存器
0x0800001E	KEYCNP	连续按键延时寄存器
0x08000020	MFC1CON	多功能计数器 1 控制寄存器
0x08000022	MFC1DATL	多功能计数器 1 计数结果低 16 位
0x08000024	MFC1DATH	多功能计数器 1 计数结果高 16 位
0x08000026	MFC2CON	多功能计数器 2 控制寄存器
0x08000028	MFC2DATL	多功能计数器 2 计数结果低 16 位
0x0800002A	MFC2DATH	多功能计数器 2 计数结果高 16 位
0x0800002C	PCP1CON	脉冲发生器 1 控制寄存器
0x0800002E	PCP1PRD	脉冲发生器 1 周期寄存器
0x08000030	PCP1CNT	脉冲发生器 1 数量寄存器
0x08000032	PCP2CON	脉冲发生器 2 控制寄存器
0x08000034	PCP2PRD	脉冲发生器 2 周期寄存器
0x08000036	PCP2CNT	脉冲发生器 2 数量寄存器
0x08000038	PCP3CON	脉冲发生器 3 控制寄存器
0x0800003A	PCP3PRD	脉冲发生器 3 周期寄存器
0x0800003C	PCP3CNT	脉冲发生器 3 数量寄存器
0x0800003E	Reserved	保留



根据表 6-2, FPGA 地 ID 寄存器在 C 语言里定义如下地宏定义:

```
#define FPGA_IDL          (*(volatile unsigned short *)0x80000000)
#define FPGA_IDH          (*(volatile unsigned short *)0x80000002)
```

这样就可以像访问 S3C44B0X 地内部寄存器一样访问 FPGA 了。通过如下函数, 就可以获得 FPGA 地 ID 寄存器里地数据了。

```
void FPGA_GetID(unsigned int *version, unsigned long *serial)
{
    unsigned long ID;
    ID=0;
    ID=FPGA_IDL;
    ID=ID|(FPGA_IDH<<16);
    *version=ID>>24;
    *serial=ID&0x00FFFFFF;
}
```

## (2) 电机运动控制

对于一般的 I/O 控制和编码器读取的操作可以使用如上所示的简单函数解决, 但是电机驱动器的控制则需要一些复杂的方法。

本系统的插补周期是 8 ms, 但是伺服周期是 4 ms, 也就是说在一次插补的过程中一般会有两次伺服的过程, 即需要向 FPGA 的驱动器控制模块发送两次数据。在第 4 章中, 已经说明了, FPGA 的驱动器控制模块有一级的缓冲能力, 在发送完脉冲信号以后产生的中断里, 不需要 CPU 立刻写入新的数值, 只要在下次中断来临之前完成写入即可。但是, 如果插补任务繁重, 有可能造成在 8 ms 之内伺服数据无法发送到 FPGA。因此, 在软件系统中添加了一个长度为 4 的 FIFO 结构, 用于存储伺服的数据, 当进入中断的时候从 FIFO 中取出一个数据发送到 FPGA。插补任务则不断检测 FIFO 内有效数据的个数, 只要 FIFO 不是满的就进行插补计算。

## 5) 硬件设备驱动程序

硬件设备驱动程序是根据系统对硬件资源的要求, 对每一个硬件功能模块编写相应的驱动软件, 即根据软硬件划分的结果, 编写硬件部分的驱动软件, 供系统应用软件调用。

硬件设备驱动包括内存分配驱动、定时器驱动、外部存储器驱动、A/D 转换驱动、LCD 驱动、键码读取驱动及通信模块驱动等。

## 2. 系统软件模块划分

嵌入式应用通常有实时性要求,即系统相当一部分功能有时间上的限制。对于实时系统来说,如果逻辑和时序出现偏差将会引起严重的后果。根据实时系统对时间的要求不同,可以划分为两种类型的实时系统:软实时系统和硬实时系统。在软实时系统中系统的宗旨是使各个任务运行得越快越好,并不要求限定某一任务必须在多长时间内完成。在硬实时系统中,各任务不仅要执行无误而且要做到准时。在实际应用中,大多数实时系统均为二者的结合。在实际的嵌入式应用中,应根据系统的功能对实时性的要求来设计嵌入式系统的应用软件,通常可以考虑以下几个方面因素。

### 1) 实时单元和分时单元的合理划分

对于实时系统,实时和分时单元的合理划分,是提高整个系统实时性能的一个重要手段。例如,对于指令处理系统来说,对指令的翻译、解释、转移、传递和应答是实时单元,而对于指令的监视打印、非法指令的打印则是分时单元。实时单元应该放到实时任务里面处理,而分时单元的处理应该由实时任务通过消息或者共享内存模式传递数据,启动分时进程在线或者后台处理。

### 2) 实时任务的划分

任务是代码运行的一个映象,从系统的角度看,任务是竞争系统资源的最小运行单元。任务可以使用或等待CPC、I/O设备以及内存空间等资源,并独立于其他任务,与它们一起并发运行(宏观上如此)。系统运行中,需要实时对任务进行调度。在没有操作系统的支持下,用户可以自己编写调度算法,来实现任务之间的切换和资源的调度。如果系统采用实时操作系统来实现,那么实时任务的调度就由操作系统来完成,操作系统内核通过一定的指示来进行任务的切换,这些指示都是来自对内核的系统调用。

在应用程序中,任务表面上具有与普通函数相似的格式,但任务有着自己较明显的特征:

- 任务具有任务初始化的起点(如获取一些系统对象的功能等)。
- 具有存放执行内容的私用数据区(如任务创建时明确定义的用户堆栈)。
- 任务的主体结构表现为一个无限循环体或有明确的中止。任务不同于函数,没有返回值。

在设计一个较为复杂的多任务应用时,进行合理的任务划分对系统的运行效率、实时性和吞吐量影响很大。任务划分过细会引起任务频繁切换的开销增加;而任务划分不够彻底,则会造成原本可以并行的操作只能串行完成,从而减少了系统的吞吐量。为了达到系



统效率和吞吐量之间的平衡与折中，在设计应用时，应遵循如下的任务分解规则（假设下述任务的发生都依赖于唯一的触发条件，如两个任务能够满足下面的条件之一，则可以合理分开）。

- 时间：两个任务所依赖的周期条件具有不同的频率和时间段。
- 异步性：两个任务所依赖的条件没有相互的时间关系。
- 优先级：两个任务所依赖的条件需要有不同的优先级。
- 清晰性/可维护性：两个任务可以在功能上或逻辑上相互分开。

从软件工程和面向对象的设计方法来看，各个模块（任务）间数据的通信量应该尽量小，并且最好少出现控制耦合（即一个任务可控制另一个任务的执行流程或功能），如果非要出现不可，则应采取相应的措施（任务间通信）使它们实现同步或互斥，以避免可能引起临界资源冲突，因为这可能造成死锁。

### 3) 提高程序效率的途径

实时程序的设计相对于分时程序的设计，尤其强调程序效率的优化。很多分时程序设计中对于提高程序效率的方法在实时程序设计中仍然有效。比较常用的优化手段有以下几种。

#### (1) 使循环体内工作量最小化

应考虑循环体内的语句是否可以放在循环体外，使循环体内工作量最小，从而提高程序的时间效率。

#### (2) 使用寄存器变量

优化关键函数时，要按照目标计算机支持的寄存器变量数，将使用的最频繁的循环变量、指针变量依次设置为寄存器变量，以提高效率。

#### (3) 使用经典高效的算法

对于一些常用的算法（例如查找、排序、hash 等），可以查找经典的库，或在原有代码的基础上改写。

#### (4) 优化函数的组织结构

对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高效率。

根据以上原则系统被分为 6 个任务，它们分别是人机界面管理、逻辑处理、运动控制模块、辅助控制、数据处理和伺服控制，这些任务都是实时周期性任务，如图 6-19 所示。操作系统选用实时操作系统，每个任务相当于操作系统的任务。伺服控制优先级最高，运动控制模块优先级次之，逻辑处理再次之，数据处理再次之，辅助控制再次之，人机界面管理优先级最低。

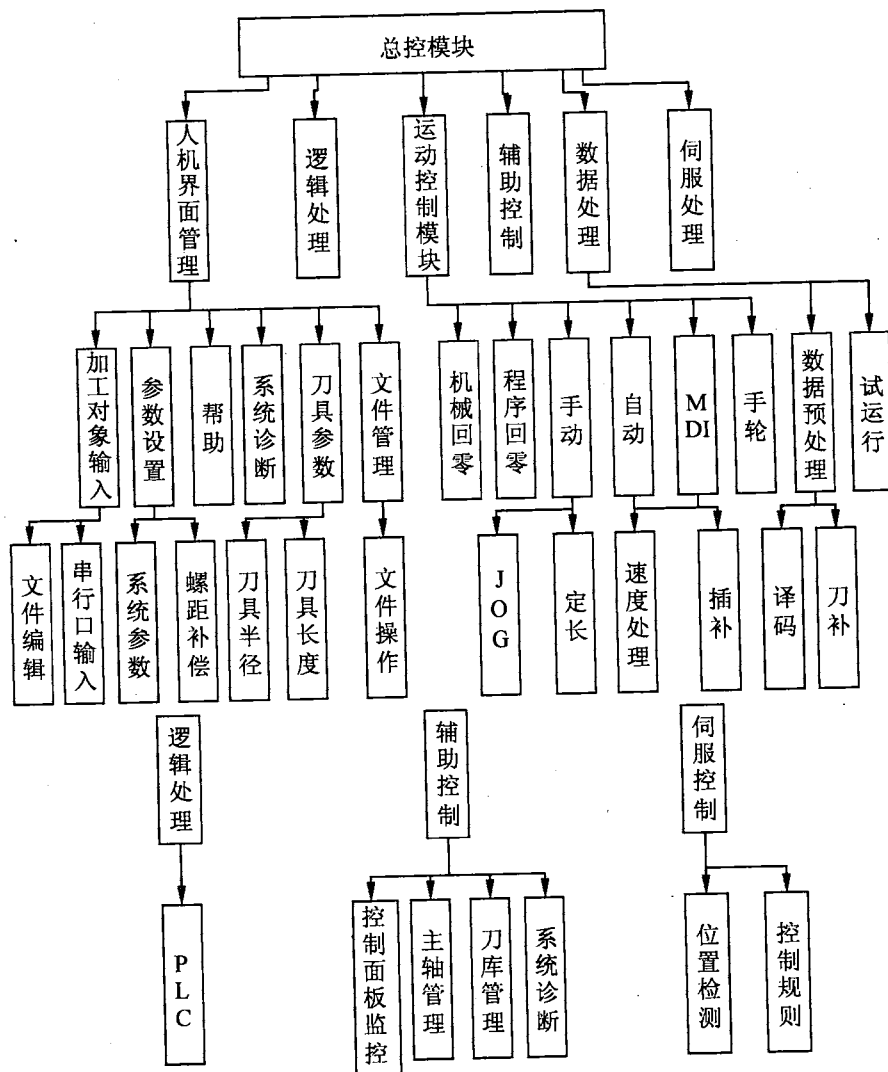


图 6-19 系统模块图

### 6.3.6 系统集成与测试

嵌入式硬件和软件组件测试完成后，它们就要被集成为部分系统或一个完整的系统。这个集成过程包括建立系统和对合成的系统的测试过程。建立系统就是把软件组件编译和连接成一个在特定目标配置上运行的可执行的二进制映像，并把其配置到特定目标机的过程。测试是要发现软件和硬件交互以及软件组件和硬件交互中的问题。这个测试包括集成

测试和系统测试。

集成测试是将已经分别通过测试的单元按设计要求组合起来再进行测试，以检查这些单元接口是否存在问题。系统测试一般由若干个不同测试组成，目的是充分运行系统，验证系统各部件能否正常工作并完成所赋予的任务。系统测试是在集成测试之后，与计算机硬件、某些支持软件、数据和人员等系统元素结合起来，在实际运行环境下对计算机系统进行严格的测试来发现软件的潜在问题，保证系统的运行。系统测试不同于功能测试。功能测试主要是验证软件功能的实现情况，不考虑各种环境及非功能问题。

集成测试应该根据系统描述来做，而且应该在一些软件和硬件一完成就开始进行。集成测试中产生的主要困难是在过程中对错误的定位。系统组件之间存在着复杂的交互行为，当一个不正常的输出被发现时，找出错误发生的源头相当困难。为了使其更容易，就必须在系统集成和测试过程中一直使用增量法。最初，集成一个最小的系统配置，然后测试这个系统。测试完毕后，一个增量一个增量地往系统中增加组件，每次增加进组件后再进行测试。

系统测试包括以下内容。

(1) 功能干涉测试

系统的单个功能在系统集成测试时已经进行了很好的测试。然而，功能之间的干涉还没有测试。这个时候就需要进行干涉测试了。功能干涉测试主要涉及到由系统提供的每个功能。测试最好的方法是建立功能干涉矩阵。

一旦定下功能测试矩阵，需要从矩阵中定下详细的测试过程。测试过程分为两类：

- 简单测试，只涉及两个功能间的干涉。
- 负载测试，涉及多个功能间的干涉测试。

对数控的自动、手动、MDI 和回零进行了测试。测试矩阵如表 6-3 所示。

表 6-3 测试矩阵

功能	自 动	手 动	MDI	回 零
自动	测试 1-1 自动运行	测试 1-2 证明自动运行情况下进入手动系统减速到零进入暂停	测试 1-3 证明自动运行情况下进入 MDI 系统执行完当前程序段停止	测试 1-4 证明自动运行情况下进入回零系统减速到零进入暂停
手动	测试 2-1 证明手动运行情况下进入自动，进入自动方式	...	...	...
MDI	测试 3-1	...	...	...
回零	测试 4-1	...	...	...

测试表明自动、手动、MDI 和回零功能没有干涉。

### (2) 压力测试

也称强度测试或负载测试。压力测试时模拟实际应用的软件环境及用户使用过程的负荷，长时间或超大负荷地运行测试软件，来测试被测系统的性能、可靠性、稳定性等。压力测试一般包括可以采用以下几种方法。

- 过载系统。
- 在实际的环境中进行负载测试。
- 负载测试时负载随时间不同而不同。
- 测试同一时间到达的负载。
- 测试具有不同服务时间的负载。
- 测试负载性能。

在测试过程中尝试让逻辑控制任务过载，运动控制任务过载，伺服控制任务过载，以及系统中断过载。以上任一任务的过载都会引起系统向伺服电机发送脉冲的间断，同时显示刷新太慢。

### (3) 容量测试

预先分析出反映软件系统应用特长的某项指标的极限量。

### (4) 性能测试

通过测试确定系统运行时的性能表现，如得到运行速度、响应时间、占有系统资源等方面的系统数据。

### (5) 安全测试

检查系统对非法侵入的防范能力。安全测试期间人员假扮非法入侵者，采用各种办法试图突破防线。

### (6) 容错测试

主要检查系统的容错能力。当系统出错时，能否在指定时间间隔内修正错误并重新启动系统。